

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

1	DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering Data Structure Index	1
1.1	DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering Data Structures . .	1
2	DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering File Index	3
2.1	DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering File List	3
3	DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering Data Structure Documentation	5
3.1	avl_node Struct Reference	5
3.2	avl_table Struct Reference	7
3.3	avl_traverser Struct Reference	9
3.4	BKConnect_ Struct Reference	11
3.5	BKConnectInfo_ Struct Reference	12
3.6	BKConnectVec_ Struct Reference	13
3.7	BKNode_ Struct Reference	15
3.8	BKNodeInfo_ Struct Reference	17
3.9	BKNodeVec_ Struct Reference	18
3.10	BKTopology_ Struct Reference	20
3.11	CPDijkNode_ Struct Reference	22
3.12	CPPrioQueue_ Struct Reference	24
3.13	CPTreeNode_ Struct Reference	26
3.14	DAMOTEConfig_ Struct Reference	28
3.15	DataBase_ Struct Reference	29
3.16	DBLabelSwitchedPath_ Struct Reference	32
3.17	DBLink_ Struct Reference	35
3.18	DBLinkState_ Struct Reference	37
3.19	DBLinkTab_ Struct Reference	40
3.20	DBLSPList_ Struct Reference	41
3.21	DBLSPVec_ Struct Reference	42

3.22 DBNode_ Struct Reference	43
3.23 DBNodeVec_ Struct Reference	44
3.24 DoubleVec_ Struct Reference	45
3.25 ErrorElem_ Struct Reference	46
3.26 ErrorList_ Struct Reference	47
3.27 libavl_allocator Struct Reference	48
3.28 LongVec_ Struct Reference	49
3.29 LSPRequest_ Struct Reference	50
3.30 LSPRequestList_ Struct Reference	53
3.31 LSPrerouteInfo_ Struct Reference	54
3.32 PredicateConfig_ Struct Reference	56
3.33 PrimaryComputationConfig_ Struct Reference	57
3.34 ReroutingConfig_ Struct Reference	59
4 DAMOTE - Decentralized Agent for MPLS Online Traffic Engineering File Documentation	61
4.1 avl.c File Reference	61
4.2 avl.h File Reference	77
4.3 backup.c File Reference	93
4.4 backup_api.h File Reference	106
4.5 backup_st.h File Reference	117
4.6 common.c File Reference	119
4.7 common.h File Reference	135
4.8 computation.c File Reference	155
4.9 computation_api.h File Reference	162
4.10 computation_st.h File Reference	169
4.11 database-oli.c File Reference	171
4.12 database.c File Reference	212
4.13 database_api.h File Reference	250
4.14 database_st.h File Reference	284
4.15 database_util.c File Reference	287
4.16 database_util.h File Reference	308
4.17 dijkstra.c File Reference	329
4.18 dijkstra.h File Reference	335
4.19 error.c File Reference	341
4.20 error.h File Reference	345
4.21 predicate.c File Reference	350
4.22 predicate.h File Reference	355

4.23	primaryPath.c File Reference	359
4.24	primaryPath_api.h File Reference	390
4.25	primaryPath_util.h File Reference	392
4.26	rerouting.c File Reference	422
4.27	rerouting.h File Reference	425
4.28	setup.c File Reference	428
4.29	setup.h File Reference	429

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:

avl_node	5
avl_table	7
avl_traverser	9
BKConnect_	11
BKConnectInfo_	12
BKConnectVec_	13
BKNode_	15
BKNodeInfo_	17
BKNodeVec_	18
BKTopology_	20
CPDijkNode_	22
CPPrioQueue_	24
CPTreeNode_	26
DAMOTEConfig_	28
DataBase_	29
DBLabelSwitchedPath_ (LSP structure)	32
DBLink_	35
DBLinkState_ (Link state structure)	37
DBLinkTab_	40
DBLSPList_	41
DBLSPVec_	42
DBNode_	43
DBNodeVec_	44
DoubleVec_	45
ErrorElem_	46
ErrorList_	47
libavl_allocator	48

LongVec_	49
LSPRequest_ (LSP Request Structure)	50
LSPRequestList_	53
LSPrerouteInfo_ (Rerouting Information structure)	54
PredicateConfig_	56
PrimaryComputationConfig_	57
ReroutingConfig_	59

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:

avl.c	61
avl.h	77
backup.c	93
backup_api.h	106
backup_st.h	117
common.c	119
common.h	135
computation.c	155
computation_api.h	162
computation_st.h	169
database-oli.c	171
database.c	212
database_api.h	250
database_st.h	284
database_util.c	287
database_util.h	308
dijkstra.c	329
dijkstra.h	335
error.c	341
error.h	345
predicate.c	350
predicate.h	355
primaryPath.c	359
primaryPath_api.h	390
primaryPath_util.h	392
rerouting.c	422
rerouting.h	425

setup.c	428
setup.h	429

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`:



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\(\)](#).

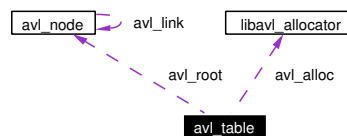
The documentation for this struct was generated from the following file:

- [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\(\)](#).

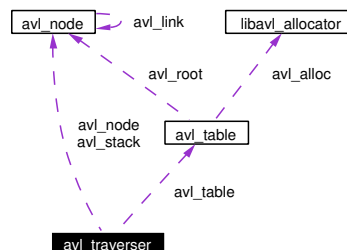
The documentation for this struct was generated from the following file:

- [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\(\)](#).

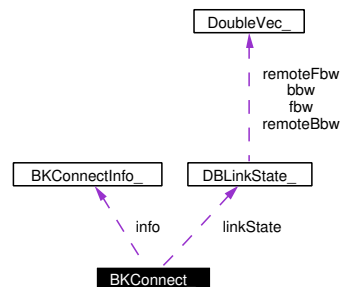
The documentation for this struct was generated from the following file:

- [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().

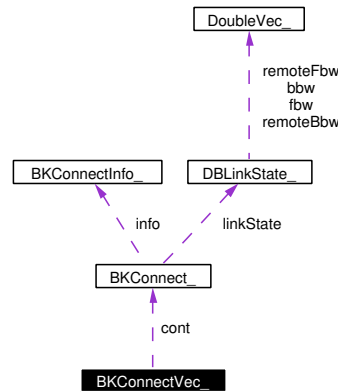
The documentation for this struct was generated from the following file:

- [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()`, `bkConnectVecSet()`, `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()`, `bkConnectVecPushBack()`, `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()`.

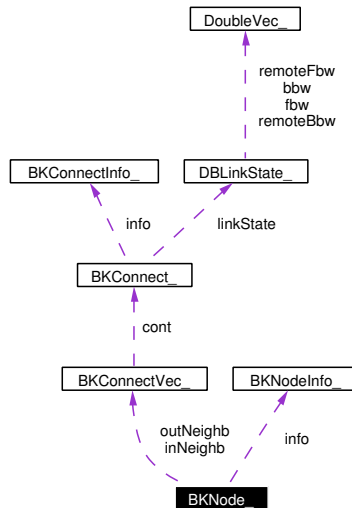
The documentation for this struct was generated from the following file:

- [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()`, `bkNodeVecPushBack()`, `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`.

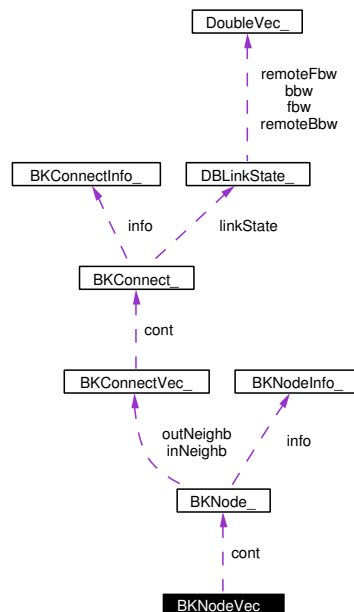
The documentation for this struct was generated from the following file:

- [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()`.

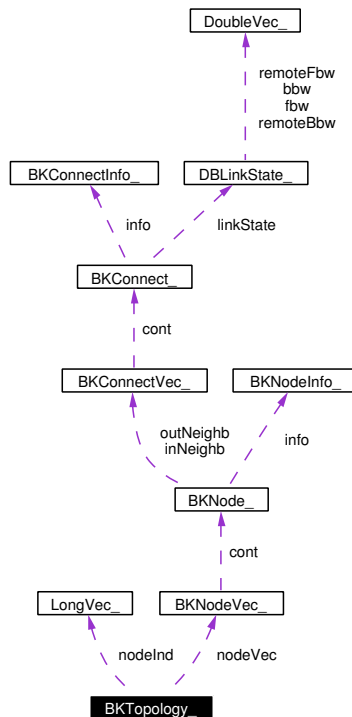
The documentation for this struct was generated from the following file:

- [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().

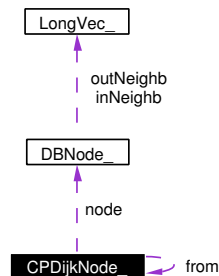
The documentation for this struct was generated from the following file:

- [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()`.

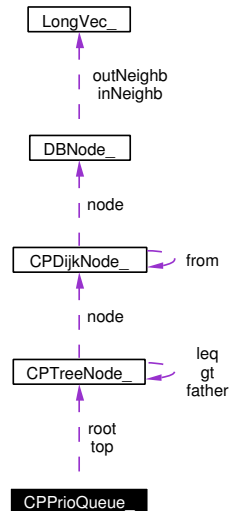
The documentation for this struct was generated from the following file:

- [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().

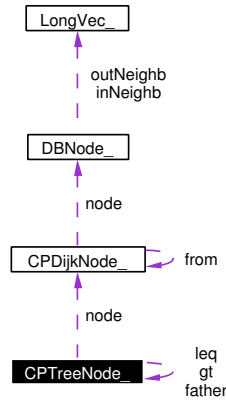
The documentation for this struct was generated from the following file:

- [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 CPopTop().

3.13.1.2 struct CPTreeNode_* CPTreeNode_::gt

Definition at line 20 of file dijkstra.h.

Referenced by CPinsertPQ(), and CPopTop().

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 CPopTop().

3.13.1.5 CPDijkNode* CPTreeNode_::node

Definition at line 22 of file dijkstra.h.

Referenced by CPinsertPQ(), and CPopTop().

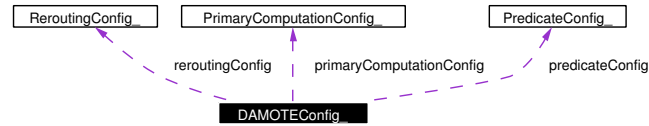
The documentation for this struct was generated from the following file:

- [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()`.

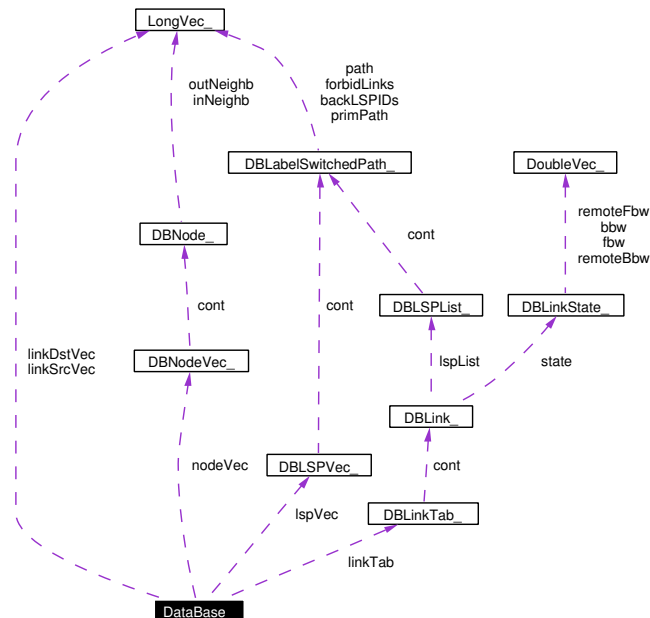
The documentation for this struct was generated from the following file:

- [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`
*Bidimensionnal 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 guarantee 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

Bidimensionnal array of Links. It is `nodeVec * nodeVec` large.

Definition at line 129 of file database_util.h.

Referenced by DBaddLink(), DBaddLSP(), DBdestroy(), DBgetLinkID(), DBgetLinkLSPs(), DBgetLinkState(), 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:

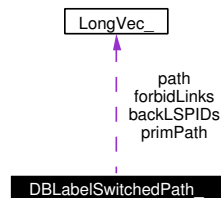
- [database_util.h](#)

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(), DBlspCopy(), 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 isValidLSPLink().

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().

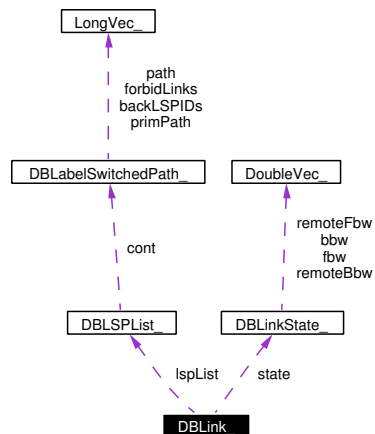
The documentation for this struct was generated from the following file:

- [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:

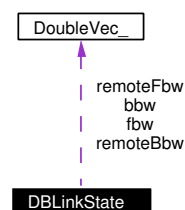
- [database_util.h](#)

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()`, `DBprintLink()`, `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()`.

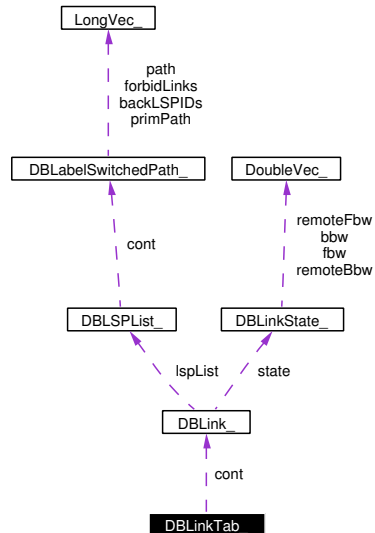
The documentation for this struct was generated from the following file:

- [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(), DBlinkTabRemove(), 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(), DBlinkTabRemove(), DBlinkTabResize(), DBlinkTabSet(), and DBprintDB().

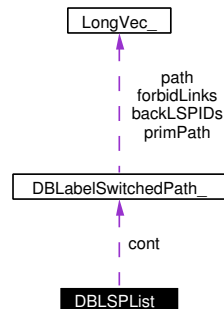
The documentation for this struct was generated from the following file:

- [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()`, `DBlspListInsert()`, `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()`, `DBlspListNew()`, `DBlspListRemove()`, and `DBprintLink()`.

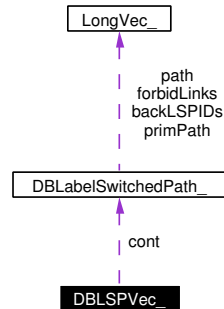
The documentation for this struct was generated from the following file:

- [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\(\)](#), [DBlspVecRemove\(\)](#), [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\(\)](#), [DBlspVecRemove\(\)](#), [DBlspVecResize\(\)](#), and [DBlspVecSet\(\)](#).

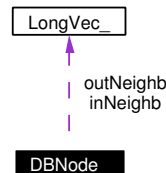
The documentation for this struct was generated from the following file:

- [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()`.

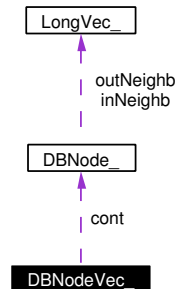
The documentation for this struct was generated from the following file:

- `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\(\)](#), [dblVecPushBack\(\)](#), 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.

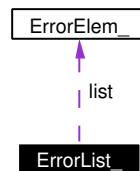
Referenced by `printErrorStack()`.

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:

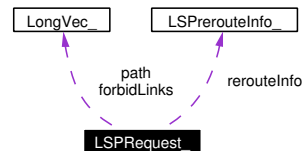
- [common.h](#)

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
- [DBLSPTType](#) [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().

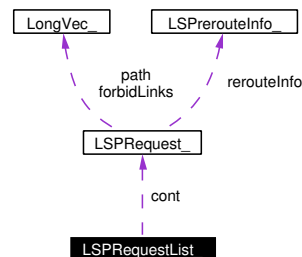
The documentation for this struct was generated from the following file:

- [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 resources of the preempted one can be used. But it is also interesting to take into account the link where the preemption occurred, 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](#)

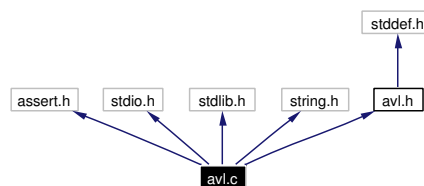
Chapter 4

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\(\)](#).

```

729 {
730     struct avl_node *stack[2 * (AVL_MAX_HEIGHT + 1)];
731     int height = 0;
732
733     struct avl_table *new;
734     const struct avl_node *x;
735     struct avl_node *y;
736
737     assert (org != NULL);
738     new = avl_create (org->avl_compare, org->avl_param,
739                     allocator != NULL ? allocator : org->avl_alloc);
740     if (new == NULL)
741         return NULL;
742     new->avl_count = org->avl_count;
743     if (new->avl_count == 0)
744         return new;
745
746     x = (const struct avl_node *) &org->avl_root;
747     y = (struct avl_node *) &new->avl_root;
748     for (;;)
749     {
750         while (x->avl_link[0] != NULL)
751             {
752                 assert (height < 2 * (AVL_MAX_HEIGHT + 1));

```

```

753
754     y->avl_link[0] =
755         new->avl_alloc->libavl_malloc (new->avl_alloc,
756                                     sizeof *y->avl_link[0]);
757     if (y->avl_link[0] == NULL)
758     {
759         if (y != (struct avl_node *) &new->avl_root)
760         {
761             y->avl_data = NULL;
762             y->avl_link[1] = NULL;
763         }
764
765         copy_error_recovery (stack, height, new, destroy);
766         return NULL;
767     }
768
769     stack[height++] = (struct avl_node *) x;
770     stack[height++] = y;
771     x = x->avl_link[0];
772     y = y->avl_link[0];
773 }
774 y->avl_link[0] = NULL;
775
776 for (;;)
777 {
778     y->avl_balance = x->avl_balance;
779     if (copy == NULL)
780         y->avl_data = x->avl_data;
781     else
782     {
783         y->avl_data = copy (x->avl_data, org->avl_param);
784         if (y->avl_data == NULL)
785         {
786             y->avl_link[1] = NULL;
787             copy_error_recovery (stack, height, new, destroy);
788             return NULL;
789         }
790     }
791
792     if (x->avl_link[1] != NULL)
793     {
794         y->avl_link[1] =
795             new->avl_alloc->libavl_malloc (new->avl_alloc,
796                                         sizeof *y->avl_link[1]);
797         if (y->avl_link[1] == NULL)
798         {
799             copy_error_recovery (stack, height, new, destroy);
800             return NULL;
801         }
802
803         x = x->avl_link[1];
804         y = y->avl_link[1];
805         break;
806     }
807     else
808         y->avl_link[1] = NULL;
809
810     if (height <= 2)
811         return new;
812
813     y = stack[--height];
814     x = stack[--height];
815 }
816 }
817 }

```

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().

```

39 {
40     struct avl_table *tree;
41
42     assert (compare != NULL);
43
44     if (allocator == NULL)
45         allocator = &avl_allocator_default;
46
47     tree = allocator->libavl_malloc (allocator, sizeof *tree);
48     if (tree == NULL)
49         return NULL;
50
51     tree->avl_root = NULL;
52     tree->avl_compare = compare;
53     tree->avl_param = param;
54     tree->avl_alloc = allocator;
55     tree->avl_count = 0;
56     tree->avl_generation = 0;
57
58     return tree;
59 }
```

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.

```

229 {
230     /* Stack of nodes. */
231     struct avl_node *pa[AVL_MAX_HEIGHT]; /* Nodes. */
232     unsigned char da[AVL_MAX_HEIGHT];    /* |avl_link[]| indexes. */
233     int k;                                /* Stack pointer. */
234
235     struct avl_node *p; /* Traverses tree to find node to delete. */
236     int cmp;            /* Result of comparison between |item| and |p|. */
237
238     assert (tree != NULL && item != NULL);
239
240     k = 0;
241     p = (struct avl_node *) &tree->avl_root;
242     for (cmp = -1; cmp != 0;
243          cmp = tree->avl_compare (item, p->avl_data, tree->avl_param))
244     {
245         int dir = cmp > 0;
246
247         pa[k] = p;
248         da[k++] = dir;
249
250         p = p->avl_link[dir];
251         if (p == NULL)
252             return NULL;
253     }
```

```

254     item = p->avl_data;
255
256     if (p->avl_link[1] == NULL)
257         pa[k - 1]->avl_link[da[k - 1]] = p->avl_link[0];
258     else
259     {
260         struct avl_node *r = p->avl_link[1];
261         if (r->avl_link[0] == NULL)
262         {
263             r->avl_link[0] = p->avl_link[0];
264             r->avl_balance = p->avl_balance;
265             pa[k - 1]->avl_link[da[k - 1]] = r;
266             da[k] = 1;
267             pa[k++] = r;
268         }
269     else
270     {
271         struct avl_node *s;
272         int j = k++;
273
274         for (;;)
275         {
276             da[k] = 0;
277             pa[k++] = r;
278             s = r->avl_link[0];
279             if (s->avl_link[0] == NULL)
280                 break;
281
282             r = s;
283         }
284
285         s->avl_link[0] = p->avl_link[0];
286         r->avl_link[0] = s->avl_link[1];
287         s->avl_link[1] = p->avl_link[1];
288         s->avl_balance = p->avl_balance;
289
290         pa[j - 1]->avl_link[da[j - 1]] = s;
291         da[j] = 1;
292         pa[j] = s;
293     }
294 }
295
296 tree->avl_alloc->libavl_free (tree->avl_alloc, p);
297
298 assert (k > 0);
299 while (--k > 0)
300 {
301     struct avl_node *y = pa[k];
302
303     if (da[k] == 0)
304     {
305         y->avl_balance++;
306         if (y->avl_balance == +1)
307             break;
308         else if (y->avl_balance == +2)
309         {
310             struct avl_node *x = y->avl_link[1];
311             if (x->avl_balance == -1)
312             {
313                 struct avl_node *w;
314                 assert (x->avl_balance == -1);
315                 w = x->avl_link[0];
316                 x->avl_link[0] = w->avl_link[1];
317                 w->avl_link[1] = x;
318                 y->avl_link[1] = w->avl_link[0];
319                 w->avl_link[0] = y;
320                 if (w->avl_balance == +1)

```

```

321         x->avl_balance = 0, y->avl_balance = -1;
322     else if (w->avl_balance == 0)
323         x->avl_balance = y->avl_balance = 0;
324     else /* |w->avl_balance == -1| */
325         x->avl_balance = +1, y->avl_balance = 0;
326     w->avl_balance = 0;
327     pa[k - 1]->avl_link[da[k - 1]] = w;
328 }
329 else
330 {
331     y->avl_link[1] = x->avl_link[0];
332     x->avl_link[0] = y;
333     pa[k - 1]->avl_link[da[k - 1]] = x;
334     if (x->avl_balance == 0)
335     {
336         x->avl_balance = -1;
337         y->avl_balance = +1;
338         break;
339     }
340     else
341         x->avl_balance = y->avl_balance = 0;
342 }
343 }
344 }
345 else
346 {
347     y->avl_balance--;
348     if (y->avl_balance == -1)
349         break;
350     else if (y->avl_balance == -2)
351     {
352         struct avl_node *x = y->avl_link[0];
353         if (x->avl_balance == +1)
354         {
355             struct avl_node *w;
356             assert (x->avl_balance == +1);
357             w = x->avl_link[1];
358             x->avl_link[1] = w->avl_link[0];
359             w->avl_link[0] = x;
360             y->avl_link[0] = w->avl_link[1];
361             w->avl_link[1] = y;
362             if (w->avl_balance == -1)
363                 x->avl_balance = 0, y->avl_balance = +1;
364             else if (w->avl_balance == 0)
365                 x->avl_balance = y->avl_balance = 0;
366             else /* |w->avl_balance == +1| */
367                 x->avl_balance = -1, y->avl_balance = 0;
368             w->avl_balance = 0;
369             pa[k - 1]->avl_link[da[k - 1]] = w;
370         }
371         else
372         {
373             y->avl_link[0] = x->avl_link[1];
374             x->avl_link[1] = y;
375             pa[k - 1]->avl_link[da[k - 1]] = x;
376             if (x->avl_balance == 0)
377             {
378                 x->avl_balance = +1;
379                 y->avl_balance = -1;
380                 break;
381             }
382             else
383                 x->avl_balance = y->avl_balance = 0;
384         }
385     }
386 }
387 }

```



```

388
389     tree->avl_count--;
390     tree->avl_generation++;
391     return (void *) item;
392 }

```

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](#).

```

823 {
824     struct avl_node *p, *q;
825
826     assert (tree != NULL);
827
828     for (p = tree->avl_root; p != NULL; p = q)
829         if (p->avl_link[0] == NULL)
830             {
831                 q = p->avl_link[1];
832                 if (destroy != NULL && p->avl_data != NULL)
833                     destroy (p->avl_data, tree->avl_param);
834                 tree->avl_alloc->libavl_free (tree->avl_alloc, p);
835             }
836         else
837             {
838                 q = p->avl_link[0];
839                 p->avl_link[0] = q->avl_link[1];
840                 q->avl_link[1] = p;
841             }
842
843     tree->avl_alloc->libavl_free (tree->avl_alloc, tree);
844 }

```

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](#).

```

65 {
66     const struct avl_node *p;
67
68     assert (tree != NULL && item != NULL);
69     for (p = tree->avl_root; p != NULL; )
70         {
71             int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
72
73             if (cmp < 0)
74                 p = p->avl_link[0];
75             else if (cmp > 0)
76                 p = p->avl_link[1];
77             else /* |cmp == 0| */
78                 return p->avl_data;
79         }
80
81     return NULL;
82 }

```

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

Definition at line 857 of file avl.c.

References free.

```

858 {
859     assert (allocator != NULL && block != NULL);
860     free (block);
861 }
```

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

Definition at line 201 of file avl.c.

References avl_probe().

```

202 {
203     void **p = avl_probe (table, item);
204     return p == NULL || *p == item ? NULL : *p;
205 }
```

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

Definition at line 849 of file avl.c.

References malloc.

```

850 {
851     assert (allocator != NULL && size > 0);
852     return malloc (size);
853 }
```

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().

```

90 {
91     struct avl_node *y, *z; /* Top node to update balance factor, and parent. */
92     struct avl_node *p, *q; /* Iterator, and parent. */
93     struct avl_node *n;     /* Newly inserted node. */
94     struct avl_node *w;     /* New root of rebalanced subtree. */
95     int dir;                /* Direction to descend. */
96
97     unsigned char da[AVL_MAX_HEIGHT]; /* Cached comparison results. */
98     int k = 0;                /* Number of cached results. */
99
100     assert (tree != NULL && item != NULL);
101
102     z = (struct avl_node *) &tree->avl_root;
103     y = tree->avl_root;
104     dir = 0;
105     for (q = z, p = y; p != NULL; q = p, p = p->avl_link[dir])
```

```

106     {
107         int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
108         if (cmp == 0)
109             return &p->avl_data;
110
111         if (p->avl_balance != 0)
112             z = q, y = p, k = 0;
113         da[k++] = dir = cmp > 0;
114     }
115
116     n = q->avl_link[dir] =
117         tree->avl_alloc->libavl_malloc (tree->avl_alloc, sizeof *n);
118     if (n == NULL)
119         return NULL;
120
121     tree->avl_count++;
122     n->avl_data = item;
123     n->avl_link[0] = n->avl_link[1] = NULL;
124     n->avl_balance = 0;
125     if (y == NULL)
126         return &n->avl_data;
127
128     for (p = y, k = 0; p != n; p = p->avl_link[da[k]], k++)
129         if (da[k] == 0)
130             p->avl_balance--;
131         else
132             p->avl_balance++;
133
134     if (y->avl_balance == -2)
135     {
136         struct avl_node *x = y->avl_link[0];
137         if (x->avl_balance == -1)
138         {
139             w = x;
140             y->avl_link[0] = x->avl_link[1];
141             x->avl_link[1] = y;
142             x->avl_balance = y->avl_balance = 0;
143         }
144         else
145         {
146             assert (x->avl_balance == +1);
147             w = x->avl_link[1];
148             x->avl_link[1] = w->avl_link[0];
149             w->avl_link[0] = x;
150             y->avl_link[0] = w->avl_link[1];
151             w->avl_link[1] = y;
152             if (w->avl_balance == -1)
153                 x->avl_balance = 0, y->avl_balance = +1;
154             else if (w->avl_balance == 0)
155                 x->avl_balance = y->avl_balance = 0;
156             else /* |w->avl_balance == +1| */
157                 x->avl_balance = -1, y->avl_balance = 0;
158             w->avl_balance = 0;
159         }
160     }
161     else if (y->avl_balance == +2)
162     {
163         struct avl_node *x = y->avl_link[1];
164         if (x->avl_balance == +1)
165         {
166             w = x;
167             y->avl_link[1] = x->avl_link[0];
168             x->avl_link[0] = y;
169             x->avl_balance = y->avl_balance = 0;
170         }
171         else
172         {

```

```

173         assert (x->avl_balance == -1);
174         w = x->avl_link[0];
175         x->avl_link[0] = w->avl_link[1];
176         w->avl_link[1] = x;
177         y->avl_link[1] = w->avl_link[0];
178         w->avl_link[0] = y;
179         if (w->avl_balance == +1)
180             x->avl_balance = 0, y->avl_balance = -1;
181         else if (w->avl_balance == 0)
182             x->avl_balance = y->avl_balance = 0;
183         else /* |w->avl_balance == -1| */
184             x->avl_balance = +1, y->avl_balance = 0;
185         w->avl_balance = 0;
186     }
187 }
188 else
189     return &n->avl_data;
190 z->avl_link[y != z->avl_link[0]] = w;
191
192 tree->avl_generation++;
193 return &n->avl_data;
194 }

```

4.1.1.10 void* avl_replace (struct [avl_table](#) * table, void * item)

Definition at line 212 of file avl.c.

References [avl_probe\(\)](#).

```

213 {
214     void **p = avl_probe (table, item);
215     if (p == NULL || *p == item)
216         return NULL;
217     else
218     {
219         void *r = *p;
220         *p = item;
221         return r;
222     }
223 }

```

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](#).

```

558 {
559     assert (trav != NULL && src != NULL);
560
561     if (trav != src)
562     {
563         trav->avl_table = src->avl_table;
564         trav->avl_node = src->avl_node;
565         trav->avl_generation = src->avl_generation;
566         if (trav->avl_generation == trav->avl_table->avl_generation)
567         {
568             trav->avl_height = src->avl_height;
569             memcpy (trav->avl_stack, (const void *) src->avl_stack,
570                 sizeof *trav->avl_stack * trav->avl_height);

```

```

571     }
572 }
573
574 return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
575 }

```

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](#).

```

686 {
687     assert (trav != NULL);
688
689     return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
690 }

```

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](#).

```

494 {
495     struct avl_node *p, *q;
496
497     assert (trav != NULL && tree != NULL && item != NULL);
498     trav->avl_table = tree;
499     trav->avl_height = 0;
500     trav->avl_generation = tree->avl_generation;
501     for (p = tree->avl_root; p != NULL; p = q)
502     {
503         int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
504
505         if (cmp < 0)
506             q = p->avl_link[0];
507         else if (cmp > 0)
508             q = p->avl_link[1];
509         else /* |cmp == 0| */
510         {
511             trav->avl_node = p;
512             return p->avl_data;
513         }
514
515         assert (trav->avl_height < AVL_MAX_HEIGHT);
516         trav->avl_stack[trav->avl_height++] = p;
517     }
518
519     trav->avl_height = 0;
520     trav->avl_node = NULL;
521     return NULL;
522 }

```

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()`.

```

438 {
439     struct avl_node *x;
440
441     assert (tree != NULL && trav != NULL);
442
443     trav->avl_table = tree;
444     trav->avl_height = 0;
445     trav->avl_generation = tree->avl_generation;
446
447     x = tree->avl_root;
448     if (x != NULL)
449         while (x->avl_link[0] != NULL)
450             {
451                 assert (trav->avl_height < AVL_MAX_HEIGHT);
452                 trav->avl_stack[trav->avl_height++] = x;
453                 x = x->avl_link[0];
454             }
455     trav->avl_node = x;
456
457     return x != NULL ? x->avl_data : NULL;
458 }
```

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()`.

```

426 {
427     trav->avl_table = tree;
428     trav->avl_node = NULL;
429     trav->avl_height = 0;
430     trav->avl_generation = tree->avl_generation;
431 }
```

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`.

```

533 {
534     void **p;
535
536     assert (trav != NULL && tree != NULL && item != NULL);
537
538     p = avl_probe (tree, item);
539     if (p != NULL)
540         {
```

```

541     trav->avl_table = tree;
542     trav->avl_node =
543         ((struct avl_node *)
544          ((char *) p - offsetof (struct avl_node, avl_data)));
545     trav->avl_generation = tree->avl_generation - 1;
546     return *p;
547 }
548 else
549 {
550     avl_t_init (trav, tree);
551     return NULL;
552 }
553 }

```

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().

```

465 {
466     struct avl_node *x;
467
468     assert (tree != NULL && trav != NULL);
469
470     trav->avl_table = tree;
471     trav->avl_height = 0;
472     trav->avl_generation = tree->avl_generation;
473
474     x = tree->avl_root;
475     if (x != NULL)
476         while (x->avl_link[1] != NULL)
477         {
478             assert (trav->avl_height < AVL_MAX_HEIGHT);
479             trav->avl_stack[trav->avl_height++] = x;
480             x = x->avl_link[1];
481         }
482     trav->avl_node = x;
483
484     return x != NULL ? x->avl_data : NULL;
485 }

```

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.

```

582 {
583     struct avl_node *x;
584
585     assert (trav != NULL);
586
587     if (trav->avl_generation != trav->avl_table->avl_generation)

```

```

588     trav_refresh (trav);
589
590     x = trav->avl_node;
591     if (x == NULL)
592     {
593         return avl_t_first (trav, trav->avl_table);
594     }
595     else if (x->avl_link[1] != NULL)
596     {
597         assert (trav->avl_height < AVL_MAX_HEIGHT);
598         trav->avl_stack[trav->avl_height++] = x;
599         x = x->avl_link[1];
600
601         while (x->avl_link[0] != NULL)
602         {
603             assert (trav->avl_height < AVL_MAX_HEIGHT);
604             trav->avl_stack[trav->avl_height++] = x;
605             x = x->avl_link[0];
606         }
607     }
608     else
609     {
610         struct avl_node *y;
611
612         do
613         {
614             if (trav->avl_height == 0)
615             {
616                 trav->avl_node = NULL;
617                 return NULL;
618             }
619
620             y = x;
621             x = trav->avl_stack[--trav->avl_height];
622         }
623         while (y == x->avl_link[1]);
624     }
625     trav->avl_node = x;
626
627     return x->avl_data;
628 }

```

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](#).

```

635 {
636     struct avl_node *x;
637
638     assert (trav != NULL);
639
640     if (trav->avl_generation != trav->avl_table->avl_generation)
641         trav_refresh (trav);
642
643     x = trav->avl_node;
644     if (x == NULL)
645     {
646         return avl_t_last (trav, trav->avl_table);
647     }
648     else if (x->avl_link[0] != NULL)

```



```

649     {
650         assert (trav->avl_height < AVL_MAX_HEIGHT);
651         trav->avl_stack[trav->avl_height++] = x;
652         x = x->avl_link[0];
653
654         while (x->avl_link[1] != NULL)
655         {
656             assert (trav->avl_height < AVL_MAX_HEIGHT);
657             trav->avl_stack[trav->avl_height++] = x;
658             x = x->avl_link[1];
659         }
660     }
661     else
662     {
663         struct avl_node *y;
664
665         do
666         {
667             if (trav->avl_height == 0)
668             {
669                 trav->avl_node = NULL;
670                 return NULL;
671             }
672
673             y = x;
674             x = trav->avl_stack[--trav->avl_height];
675         }
676         while (y == x->avl_link[0]);
677     }
678     trav->avl_node = x;
679
680     return x->avl_data;
681 }

```

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](#).

```

697 {
698     void *old;
699
700     assert (trav != NULL && trav->avl_node != NULL && new != NULL);
701     old = trav->avl_node->avl_data;
702     trav->avl_node->avl_data = new;
703     return old;
704 }

```

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`.

```
885 {  
886     void *p = avl_delete (table, item);  
887     assert (p != NULL);  
888     return p;  
889 }
```

4.1.2.3 void(avl_assert_insert)(struct [avl_table](#) *table, void *item)

Definition at line 875 of file `avl.c`.

```
876 {  
877     void **p = avl_probe (table, item);  
878     assert (p != NULL && *p == item);  
879 }
```

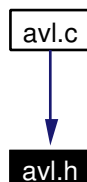
4.2 avl.h File Reference

```
#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\(\)](#).

```

729 {
730     struct avl_node *stack[2 * (AVL_MAX_HEIGHT + 1)];
731     int height = 0;
732
733     struct avl_table *new;
734     const struct avl_node *x;
735     struct avl_node *y;
736
737     assert (org != NULL);
738     new = avl_create (org->avl_compare, org->avl_param,
739                     allocator != NULL ? allocator : org->avl_alloc);
740     if (new == NULL)
741         return NULL;
742     new->avl_count = org->avl_count;
743     if (new->avl_count == 0)
744         return new;
745
746     x = (const struct avl_node *) &org->avl_root;
747     y = (struct avl_node *) &new->avl_root;
748     for (;;)
749     {
750         while (x->avl_link[0] != NULL)
751         {
752             assert (height < 2 * (AVL_MAX_HEIGHT + 1));
753
754             y->avl_link[0] =
755                 new->avl_alloc->libavl_malloc (new->avl_alloc,
756                                                sizeof *y->avl_link[0]);
757             if (y->avl_link[0] == NULL)
758             {
759                 if (y != (struct avl_node *) &new->avl_root)

```

```

760         {
761             y->avl_data = NULL;
762             y->avl_link[1] = NULL;
763         }
764
765         copy_error_recovery (stack, height, new, destroy);
766         return NULL;
767     }
768
769     stack[height++] = (struct avl_node *) x;
770     stack[height++] = y;
771     x = x->avl_link[0];
772     y = y->avl_link[0];
773 }
774 y->avl_link[0] = NULL;
775
776 for (;;)
777 {
778     y->avl_balance = x->avl_balance;
779     if (copy == NULL)
780         y->avl_data = x->avl_data;
781     else
782     {
783         y->avl_data = copy (x->avl_data, org->avl_param);
784         if (y->avl_data == NULL)
785         {
786             y->avl_link[1] = NULL;
787             copy_error_recovery (stack, height, new, destroy);
788             return NULL;
789         }
790     }
791
792     if (x->avl_link[1] != NULL)
793     {
794         y->avl_link[1] =
795             new->avl_alloc->libavl_malloc (new->avl_alloc,
796                                           sizeof *y->avl_link[1]);
797         if (y->avl_link[1] == NULL)
798         {
799             copy_error_recovery (stack, height, new, destroy);
800             return NULL;
801         }
802
803         x = x->avl_link[1];
804         y = y->avl_link[1];
805         break;
806     }
807     else
808         y->avl_link[1] = NULL;
809
810     if (height <= 2)
811         return new;
812
813     y = stack[--height];
814     x = stack[--height];
815 }
816 }
817 }

```

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()`.

```

39 {
40     struct avl_table *tree;
41
42     assert (compare != NULL);
43
44     if (allocator == NULL)
45         allocator = &avl_allocator_default;
46
47     tree = allocator->libavl_malloc (allocator, sizeof *tree);
48     if (tree == NULL)
49         return NULL;
50
51     tree->avl_root = NULL;
52     tree->avl_compare = compare;
53     tree->avl_param = param;
54     tree->avl_alloc = allocator;
55     tree->avl_count = 0;
56     tree->avl_generation = 0;
57
58     return tree;
59 }
```

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`.

```

229 {
230     /* Stack of nodes. */
231     struct avl_node *pa[AVL_MAX_HEIGHT]; /* Nodes. */
232     unsigned char da[AVL_MAX_HEIGHT];    /* |avl_link[]| indexes. */
233     int k;                                /* Stack pointer. */
234
235     struct avl_node *p; /* Traverses tree to find node to delete. */
236     int cmp;            /* Result of comparison between |item| and |p|. */
237
238     assert (tree != NULL && item != NULL);
239
240     k = 0;
241     p = (struct avl_node *) &tree->avl_root;
242     for (cmp = -1; cmp != 0;
243          cmp = tree->avl_compare (item, p->avl_data, tree->avl_param))
244     {
245         int dir = cmp > 0;
246
247         pa[k] = p;
248         da[k++] = dir;
249
250         p = p->avl_link[dir];
251         if (p == NULL)
252             return NULL;
253     }
254     item = p->avl_data;
255
256     if (p->avl_link[1] == NULL)
257         pa[k - 1]->avl_link[da[k - 1]] = p->avl_link[0];
258     else
259     {
260         struct avl_node *r = p->avl_link[1];
261         if (r->avl_link[0] == NULL)
262             {
```

```

263     r->avl_link[0] = p->avl_link[0];
264     r->avl_balance = p->avl_balance;
265     pa[k - 1]->avl_link[da[k - 1]] = r;
266     da[k] = 1;
267     pa[k++] = r;
268 }
269 else
270 {
271     struct avl_node *s;
272     int j = k++;
273
274     for (;;)
275     {
276         da[k] = 0;
277         pa[k++] = r;
278         s = r->avl_link[0];
279         if (s->avl_link[0] == NULL)
280             break;
281
282         r = s;
283     }
284
285     s->avl_link[0] = p->avl_link[0];
286     r->avl_link[0] = s->avl_link[1];
287     s->avl_link[1] = p->avl_link[1];
288     s->avl_balance = p->avl_balance;
289
290     pa[j - 1]->avl_link[da[j - 1]] = s;
291     da[j] = 1;
292     pa[j] = s;
293 }
294 }
295
296 tree->avl_alloc->libavl_free (tree->avl_alloc, p);
297
298 assert (k > 0);
299 while (--k > 0)
300 {
301     struct avl_node *y = pa[k];
302
303     if (da[k] == 0)
304     {
305         y->avl_balance++;
306         if (y->avl_balance == +1)
307             break;
308         else if (y->avl_balance == +2)
309         {
310             struct avl_node *x = y->avl_link[1];
311             if (x->avl_balance == -1)
312             {
313                 struct avl_node *w;
314                 assert (x->avl_balance == -1);
315                 w = x->avl_link[0];
316                 x->avl_link[0] = w->avl_link[1];
317                 w->avl_link[1] = x;
318                 y->avl_link[1] = w->avl_link[0];
319                 w->avl_link[0] = y;
320                 if (w->avl_balance == +1)
321                     x->avl_balance = 0, y->avl_balance = -1;
322                 else if (w->avl_balance == 0)
323                     x->avl_balance = y->avl_balance = 0;
324                 else /* |w->avl_balance == -1| */
325                     x->avl_balance = +1, y->avl_balance = 0;
326                 w->avl_balance = 0;
327                 pa[k - 1]->avl_link[da[k - 1]] = w;
328             }
329             else

```



```

330         {
331             y->avl_link[1] = x->avl_link[0];
332             x->avl_link[0] = y;
333             pa[k - 1]->avl_link[da[k - 1]] = x;
334             if (x->avl_balance == 0)
335             {
336                 x->avl_balance = -1;
337                 y->avl_balance = +1;
338                 break;
339             }
340             else
341                 x->avl_balance = y->avl_balance = 0;
342         }
343     }
344 }
345 else
346 {
347     y->avl_balance--;
348     if (y->avl_balance == -1)
349         break;
350     else if (y->avl_balance == -2)
351     {
352         struct avl_node *x = y->avl_link[0];
353         if (x->avl_balance == +1)
354         {
355             struct avl_node *w;
356             assert (x->avl_balance == +1);
357             w = x->avl_link[1];
358             x->avl_link[1] = w->avl_link[0];
359             w->avl_link[0] = x;
360             y->avl_link[0] = w->avl_link[1];
361             w->avl_link[1] = y;
362             if (w->avl_balance == -1)
363                 x->avl_balance = 0, y->avl_balance = +1;
364             else if (w->avl_balance == 0)
365                 x->avl_balance = y->avl_balance = 0;
366             else /* |w->avl_balance == +1| */
367                 x->avl_balance = -1, y->avl_balance = 0;
368             w->avl_balance = 0;
369             pa[k - 1]->avl_link[da[k - 1]] = w;
370         }
371         else
372         {
373             y->avl_link[0] = x->avl_link[1];
374             x->avl_link[1] = y;
375             pa[k - 1]->avl_link[da[k - 1]] = x;
376             if (x->avl_balance == 0)
377             {
378                 x->avl_balance = +1;
379                 y->avl_balance = -1;
380                 break;
381             }
382             else
383                 x->avl_balance = y->avl_balance = 0;
384         }
385     }
386 }
387 }
388
389 tree->avl_count--;
390 tree->avl_generation++;
391 return (void *) item;
392 }

```

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.

```

823 {
824     struct avl_node *p, *q;
825
826     assert (tree != NULL);
827
828     for (p = tree->avl_root; p != NULL; p = q)
829         if (p->avl_link[0] == NULL)
830             {
831                 q = p->avl_link[1];
832                 if (destroy != NULL && p->avl_data != NULL)
833                     destroy (p->avl_data, tree->avl_param);
834                 tree->avl_alloc->libavl_free (tree->avl_alloc, p);
835             }
836         else
837             {
838                 q = p->avl_link[0];
839                 p->avl_link[0] = q->avl_link[1];
840                 q->avl_link[1] = p;
841             }
842
843     tree->avl_alloc->libavl_free (tree->avl_alloc, tree);
844 }
```

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.

```

65 {
66     const struct avl_node *p;
67
68     assert (tree != NULL && item != NULL);
69     for (p = tree->avl_root; p != NULL; )
70         {
71             int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
72
73             if (cmp < 0)
74                 p = p->avl_link[0];
75             else if (cmp > 0)
76                 p = p->avl_link[1];
77             else /* |cmp| == 0 */
78                 return p->avl_data;
79         }
80
81     return NULL;
82 }
```

4.2.3.8 void avl_free (struct libavl_allocator *, void *)

Definition at line 857 of file avl.c.

References free.

```

858 {
859     assert (allocator != NULL && block != NULL);
860     free (block);
861 }

```

4.2.3.9 void* avl_insert (struct [avl_table](#) *, void *)

Definition at line 201 of file avl.c.

References [avl_probe\(\)](#).

```

202 {
203     void **p = avl_probe (table, item);
204     return p == NULL || *p == item ? NULL : *p;
205 }

```

4.2.3.10 void* avl_malloc (struct [libavl_allocator](#) *, size_t)

Definition at line 849 of file avl.c.

References [malloc](#).

```

850 {
851     assert (allocator != NULL && size > 0);
852     return malloc (size);
853 }

```

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\(\)](#).

```

90 {
91     struct avl_node *y, *z; /* Top node to update balance factor, and parent. */
92     struct avl_node *p, *q; /* Iterator, and parent. */
93     struct avl_node *n; /* Newly inserted node. */
94     struct avl_node *w; /* New root of rebalanced subtree. */
95     int dir; /* Direction to descend. */
96
97     unsigned char da[AVL_MAX_HEIGHT]; /* Cached comparison results. */
98     int k = 0; /* Number of cached results. */
99
100     assert (tree != NULL && item != NULL);
101
102     z = (struct avl_node *) &tree->avl_root;
103     y = tree->avl_root;
104     dir = 0;
105     for (q = z, p = y; p != NULL; q = p, p = p->avl_link[dir])
106     {
107         int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
108         if (cmp == 0)
109             return &p->avl_data;
110
111         if (p->avl_balance != 0)
112             z = q, y = p, k = 0;
113     }

```

```

113     da[k++] = dir = cmp > 0;
114 }
115
116 n = q->avl_link[dir] =
117     tree->avl_alloc->libavl_malloc (tree->avl_alloc, sizeof *n);
118 if (n == NULL)
119     return NULL;
120
121 tree->avl_count++;
122 n->avl_data = item;
123 n->avl_link[0] = n->avl_link[1] = NULL;
124 n->avl_balance = 0;
125 if (y == NULL)
126     return &n->avl_data;
127
128 for (p = y, k = 0; p != n; p = p->avl_link[da[k]], k++)
129     if (da[k] == 0)
130         p->avl_balance--;
131     else
132         p->avl_balance++;
133
134 if (y->avl_balance == -2)
135 {
136     struct avl_node *x = y->avl_link[0];
137     if (x->avl_balance == -1)
138     {
139         w = x;
140         y->avl_link[0] = x->avl_link[1];
141         x->avl_link[1] = y;
142         x->avl_balance = y->avl_balance = 0;
143     }
144     else
145     {
146         assert (x->avl_balance == +1);
147         w = x->avl_link[1];
148         x->avl_link[1] = w->avl_link[0];
149         w->avl_link[0] = x;
150         y->avl_link[0] = w->avl_link[1];
151         w->avl_link[1] = y;
152         if (w->avl_balance == -1)
153             x->avl_balance = 0, y->avl_balance = +1;
154         else if (w->avl_balance == 0)
155             x->avl_balance = y->avl_balance = 0;
156         else /* |w->avl_balance == +1| */
157             x->avl_balance = -1, y->avl_balance = 0;
158         w->avl_balance = 0;
159     }
160 }
161 else if (y->avl_balance == +2)
162 {
163     struct avl_node *x = y->avl_link[1];
164     if (x->avl_balance == +1)
165     {
166         w = x;
167         y->avl_link[1] = x->avl_link[0];
168         x->avl_link[0] = y;
169         x->avl_balance = y->avl_balance = 0;
170     }
171     else
172     {
173         assert (x->avl_balance == -1);
174         w = x->avl_link[0];
175         x->avl_link[0] = w->avl_link[1];
176         w->avl_link[1] = x;
177         y->avl_link[1] = w->avl_link[0];
178         w->avl_link[0] = y;
179         if (w->avl_balance == +1)

```

```

180         x->avl_balance = 0, y->avl_balance = -1;
181     else if (w->avl_balance == 0)
182         x->avl_balance = y->avl_balance = 0;
183     else /* |w->avl_balance == -1| */
184         x->avl_balance = +1, y->avl_balance = 0;
185     w->avl_balance = 0;
186 }
187 }
188 else
189     return &n->avl_data;
190 z->avl_link[y != z->avl_link[0]] = w;
191
192 tree->avl_generation++;
193 return &n->avl_data;
194 }

```

4.2.3.12 void* avl_replace (struct [avl_table](#) *, void *)

Definition at line 212 of file avl.c.

References [avl_probe\(\)](#).

```

213 {
214     void **p = avl_probe (table, item);
215     if (p == NULL || *p == item)
216         return NULL;
217     else
218     {
219         void *r = *p;
220         *p = item;
221         return r;
222     }
223 }

```

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](#).

```

558 {
559     assert (trav != NULL && src != NULL);
560
561     if (trav != src)
562     {
563         trav->avl_table = src->avl_table;
564         trav->avl_node = src->avl_node;
565         trav->avl_generation = src->avl_generation;
566         if (trav->avl_generation == trav->avl_table->avl_generation)
567         {
568             trav->avl_height = src->avl_height;
569             memcpy (trav->avl_stack, (const void *) src->avl_stack,
570                 sizeof *trav->avl_stack * trav->avl_height);
571         }
572     }
573
574     return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
575 }

```

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.

```

686 {
687     assert (trav != NULL);
688
689     return trav->avl_node != NULL ? trav->avl_node->avl_data : NULL;
690 }
```

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.

```

494 {
495     struct avl_node *p, *q;
496
497     assert (trav != NULL && tree != NULL && item != NULL);
498     trav->avl_table = tree;
499     trav->avl_height = 0;
500     trav->avl_generation = tree->avl_generation;
501     for (p = tree->avl_root; p != NULL; p = q)
502     {
503         int cmp = tree->avl_compare (item, p->avl_data, tree->avl_param);
504
505         if (cmp < 0)
506             q = p->avl_link[0];
507         else if (cmp > 0)
508             q = p->avl_link[1];
509         else /* |cmp == 0| */
510         {
511             trav->avl_node = p;
512             return p->avl_data;
513         }
514
515         assert (trav->avl_height < AVL_MAX_HEIGHT);
516         trav->avl_stack[trav->avl_height++] = p;
517     }
518
519     trav->avl_height = 0;
520     trav->avl_node = NULL;
521     return NULL;
522 }
```

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().

```

438 {
439     struct avl_node *x;
440
441     assert (tree != NULL && trav != NULL);
442
443     trav->avl_table = tree;
444     trav->avl_height = 0;
445     trav->avl_generation = tree->avl_generation;
446
447     x = tree->avl_root;
448     if (x != NULL)
449         while (x->avl_link[0] != NULL)
450             {
451                 assert (trav->avl_height < AVL_MAX_HEIGHT);
452                 trav->avl_stack[trav->avl_height++] = x;
453                 x = x->avl_link[0];
454             }
455     trav->avl_node = x;
456
457     return x != NULL ? x->avl_data : NULL;
458 }

```

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().

```

426 {
427     trav->avl_table = tree;
428     trav->avl_node = NULL;
429     trav->avl_height = 0;
430     trav->avl_generation = tree->avl_generation;
431 }

```

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.

```

533 {
534     void **p;
535
536     assert (trav != NULL && tree != NULL && item != NULL);
537
538     p = avl_probe (tree, item);
539     if (p != NULL)
540     {
541         trav->avl_table = tree;
542         trav->avl_node =
543             ((struct avl_node *)
544              ((char *) p - offsetof (struct avl_node, avl_data)));
545         trav->avl_generation = tree->avl_generation - 1;
546         return *p;
547     }
548     else

```

```

549     {
550         avl_t_init (trav, tree);
551         return NULL;
552     }
553 }

```

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().

```

465 {
466     struct avl_node *x;
467
468     assert (tree != NULL && trav != NULL);
469
470     trav->avl_table = tree;
471     trav->avl_height = 0;
472     trav->avl_generation = tree->avl_generation;
473
474     x = tree->avl_root;
475     if (x != NULL)
476         while (x->avl_link[1] != NULL)
477             {
478                 assert (trav->avl_height < AVL_MAX_HEIGHT);
479                 trav->avl_stack[trav->avl_height++] = x;
480                 x = x->avl_link[1];
481             }
482     trav->avl_node = x;
483
484     return x != NULL ? x->avl_data : NULL;
485 }

```

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.

```

582 {
583     struct avl_node *x;
584
585     assert (trav != NULL);
586
587     if (trav->avl_generation != trav->avl_table->avl_generation)
588         trav_refresh (trav);
589
590     x = trav->avl_node;
591     if (x == NULL)
592     {
593         return avl_t_first (trav, trav->avl_table);
594     }
595     else if (x->avl_link[1] != NULL)

```



```

596     {
597         assert (trav->avl_height < AVL_MAX_HEIGHT);
598         trav->avl_stack[trav->avl_height++] = x;
599         x = x->avl_link[1];
600
601         while (x->avl_link[0] != NULL)
602         {
603             assert (trav->avl_height < AVL_MAX_HEIGHT);
604             trav->avl_stack[trav->avl_height++] = x;
605             x = x->avl_link[0];
606         }
607     }
608     else
609     {
610         struct avl_node *y;
611
612         do
613         {
614             if (trav->avl_height == 0)
615             {
616                 trav->avl_node = NULL;
617                 return NULL;
618             }
619
620             y = x;
621             x = trav->avl_stack[--trav->avl_height];
622         }
623         while (y == x->avl_link[1]);
624     }
625     trav->avl_node = x;
626
627     return x->avl_data;
628 }

```

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](#).

```

635 {
636     struct avl_node *x;
637
638     assert (trav != NULL);
639
640     if (trav->avl_generation != trav->avl_table->avl_generation)
641         trav_refresh (trav);
642
643     x = trav->avl_node;
644     if (x == NULL)
645     {
646         return avl_t_last (trav, trav->avl_table);
647     }
648     else if (x->avl_link[0] != NULL)
649     {
650         assert (trav->avl_height < AVL_MAX_HEIGHT);
651         trav->avl_stack[trav->avl_height++] = x;
652         x = x->avl_link[0];
653
654         while (x->avl_link[1] != NULL)
655         {
656             assert (trav->avl_height < AVL_MAX_HEIGHT);

```

```

657         trav->avl_stack[trav->avl_height++] = x;
658         x = x->avl_link[1];
659     }
660 }
661 else
662 {
663     struct avl_node *y;
664
665     do
666     {
667         if (trav->avl_height == 0)
668         {
669             trav->avl_node = NULL;
670             return NULL;
671         }
672
673         y = x;
674         x = trav->avl_stack[--trav->avl_height];
675     }
676     while (y == x->avl_link[0]);
677 }
678 trav->avl_node = x;
679
680 return x->avl_data;
681 }

```

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](#).

```

697 {
698     void *old;
699
700     assert (trav != NULL && trav->avl_node != NULL && new != NULL);
701     old = trav->avl_node->avl_data;
702     trav->avl_node->avl_data = new;
703     return old;
704 }

```

4.2.4 Variable Documentation

4.2.4.1 struct [libavl_allocator](#) [avl_allocator_default](#)

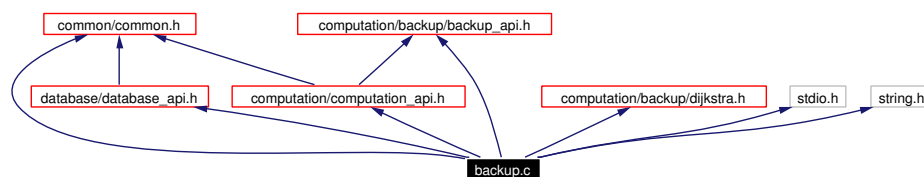
Definition at line 48 of file avl.h.

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:



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(), DBlinkStateDestroy(), DBlinkStateNew(), FALSE, free, CPDijkNode::from, GLOBAL, GLOBAL_BACK, DBLabelSwitchedPath::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 WARNING.

```

147 {
148     long i,j,pNodeIndex,start,pNode;
149     DBLinkState *ls, *oldLS;
150     DBLinkState** lsList;
151     CPDijkNode *dn=NULL;
152     CPDijkNode** nodeList;
153     bool reachPrimary;
154     CPPrioQueue toBeTreated;
155     LongList* neigh;
156     LongList forbiddenLinks;
157     DBLabelSwitchedPath* lsp;
158     LSPRequest* req=NULL;
159     double newVal;
160     int pType=0;
161     int src, dst;
162
163     enum {NODE_FAILURE, LINK_FAILURE};
164
165 #if defined LINUX && defined TIMING && defined TIME3
166     struct timezone tz;
167     struct timeval t1,t2;
168 #endif
169
170     if (dataBase == NULL || reqList == NULL)
171     {
172         addError(CRITICAL,"Wrong argument in %s at line %d",
173             __FILE__,__LINE__);
174         return -1;
175     }
176
177     if (lspID < 0 || ((lsp = DBgetLSP(dataBase, lspID)) == NULL))
178     {
179         addError(CRITICAL,"Cannot find lsp ID in %s at line %d",
180             __FILE__,__LINE__);
181         return -1;
182     }
183
184     if ((lsList = calloc(dataBase->linkSrcVec.top, sizeof(DBLinkState*))) == NULL)
185     {
186         addError(CRITICAL,"Cannot allocate a required structure in %s at line %d",
187             __FILE__,__LINE__);
188         return -1;
189     }
190
191     if ((nodeList = calloc(dataBase->nodeVec.top, sizeof(CPDijkNode*))) == NULL)
192     {
193         addError(CRITICAL,"Cannot allocate a required structure in %s at line %d",
194             __FILE__,__LINE__);
195         free(lsList);
196         return -1;
197     }
198
199     // duplicate all the link-states
200     for (i=0; i<dataBase->linkSrcVec.top; i++)
201     {
202         src = DBgetLinkSrc(dataBase, i);

```

```

203     dst = DBgetLinkDst(dataBase, i);
204
205     if (src != -1 && dst != -1)
206     {
207         if ((oldLS = DBgetLinkState(dataBase, src, dst)) == NULL)
208         {
209             addError(WARNING, "Oups there should be a link-state here in %s at line %d",
210                     __FILE__, __LINE__);
211             continue;
212         }
213
214         if ((ls = DBlinkStateNew()) == NULL)
215         {
216             addError(CRITICAL, "Cannot duplicate all the link-states in %s at line %d",
217                     __FILE__, __LINE__);
218             continue;
219         }
220
221         if (DBlinkStateCopy(ls, oldLS) < 0)
222         {
223             addError(CRITICAL, "Something went wrong while copying in %s at line %d",
224                     __FILE__, __LINE__);
225             DBlinkStateDestroy(ls);
226             continue;
227         }
228
229         lsList[i] = ls;
230     }
231     else
232     {
233         addError(INFO, "Warning there is no link numbered %ld : src = %ld, dst = %ld .... in %s at
234             __FILE__, __LINE__);
235     }
236 }
237
238 // create the Dijk Nodes ... used for the computation
239 for (i=0; i<dataBase->nodeVec.top; i++)
240 {
241     if (dataBase->nodeVec.cont[i] != NULL)
242     {
243         if ((dn = calloc(1, sizeof(CPDijkNode))) == NULL)
244         {
245             addError(CRITICAL, "Cannot create the Dijk nodes in %s at line %d",
246                     __FILE__, __LINE__);
247             continue;
248         }
249
250         dn->node = dataBase->nodeVec.cont[i];
251         dn->val = -1;
252         dn->marked = FALSE;
253
254         nodeList[i] = dn;
255     }
256 }
257
258 printf("Primary path :");
259 for (i=0; i<lsp->path.top - 1; ++i)
260 {
261     printf("%ld - ", lsp->path.cont[i]);
262 }
263 printf("%ld\n", lsp->path.cont[i]);
264
265 // now start the calculation ....
266
267 #if defined LINUX && defined TIMING && defined TIME3
268     gettimeofday(&t1, &tz);
269 #endif

```

```

270
271 // init a PrioQueue
272 CPinitPQ(&toBeTreated);
273
274 // init the forbiddenLinks;
275 longListInit(&forbiddenLinks, -1);
276
277 if (type == LOCAL)
278 {
279     // init the list of request to return;
280     lspRequestListResize(reqList, lsp->path.top-1);
281     for (i=0; i<lsp->path.top-1; ++i)
282     {
283         req = lspRequestListGet(reqList, i);
284
285         req->primID = lsp->id;
286         req->type = LOCAL_BACK;
287         if (lsp->precedence + 1 < NB_PREEMPTION)
288             req->precedence = lsp->precedence + 1;
289         else
290             req->precedence = lsp->precedence;
291         memmove(&(req->bw), &(lsp->bw), NB_OA * sizeof(double));
292     }
293
294     // for (pNodeIndex=1; pNodeIndex<req->path.top; ++pNodeIndex) // start at 1 because we cannot
295     for (pNodeIndex=lsp->path.top - 1; pNodeIndex>0; --pNodeIndex)
296     {
297         pNode = lsp->path.cont[pNodeIndex];
298         start = lsp->path.cont[pNodeIndex - 1];
299
300         pType = NODE_FAILURE;
301
302         // mark the forbidden links
303         forbiddenLinks.top = 0;
304
305         if (pType == NODE_FAILURE)
306         {
307             neigh = DBgetNodeInNeighb(dataBase, pNode);
308             if (neigh == NULL)
309             {
310                 addError(CRITICAL, "The protected node must have some neighbour in %s at line %d",
311                     __FILE__, __LINE__);
312                 return -1;
313             }
314
315             for (i=0; i<neigh->top; ++i)
316             {
317                 longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[i], pNode));
318             }
319         }
320         else if (pType == LINK_FAILURE)
321         {
322             longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));
323         }
324
325         // clear the PQ;
326         while (CPpopTop(&toBeTreated) != NULL);
327
328         // clear all the marks
329         for (i=0; i<dataBase->nodeVec.top; i++)
330         {
331             nodeList[i]->marked = FALSE;
332             nodeList[i]->val = -1;
333             nodeList[i]->from = NULL;
334         }
335
336         // push the first node on the PQ

```

```

337         CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
338
339     reachPrimary = FALSE;
340     while (reachPrimary == FALSE)
341     {
342         if ((dn = CPpopTop(&toBeTreated)) == NULL)
343         {
344             // Oups ... impossible to reach the primary
345             // if we are in node protection mode, switch back to link protection
346             if (pType == NODE_FAILURE)
347             {
348                 pType = LINK_FAILURE;
349
350                 // mark the forbidden links
351                 forbiddenLinks.top = 0;
352                 longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));
353
354                 // clear all marked nodes
355                 for (i=0; i<dataBase->nodeVec.top; i++)
356                 {
357                     nodeList[i]->marked = FALSE;
358                     nodeList[i]->val = -1;
359                     nodeList[i]->from = NULL;
360                 }
361
362                 // push the first node on the PQ
363                 CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
364
365                 // re-enter the loop
366                 continue;
367             }
368             else
369             {
370                 break;
371             }
372         }
373
374         // as we don't remove marked node immediatelly we may encounter one now so we skip it
375         if (dn->marked == TRUE)
376             continue;
377
378         // mark the node
379         dn->marked = TRUE;
380
381         // check the stop condition
382         for (i=pNodeIndex; i<lsp->path.top; ++i)
383             if (lsp->path.cont[i] == dn->node->id)
384             {
385                 reachPrimary = TRUE;
386                 break;
387             }
388
389         // we have finished ... leave the while loop
390         if (reachPrimary == TRUE)
391             break;
392
393         // find the neighbours
394         neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);
395
396         if (neigh != NULL)
397         {
398             for (i=0; i<neigh->top; ++i)
399             {
400                 int id;
401                 double cost;
402
403                 // check if the node is not already marked

```

```

404         if (nodeList[neigh->cont[i]]->marked == TRUE)
405             continue;
406
407         // check if the link is valid
408         id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
409         for (j=0; j<forbiddenLinks.top; ++j)
410             if (forbiddenLinks.cont[j] == id)
411                 break;
412
413         if (j != forbiddenLinks.top)
414             continue;
415
416         // ok now update the node ...
417         cost = computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn, lsp, ty
418         if (cost >= 0) {
419             newVal = dn->val + cost;
420
421             if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeLis
422                 {
423                     nodeList[neigh->cont[i]]->val = newVal;
424                     nodeList[neigh->cont[i]]->from = dn;
425                     CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
426                 }
427             }
428         }
429     }
430 }
431
432 if (reachPrimary == TRUE)
433 {
434     req = lspRequestListGet(reqList, pNodeIndex-1);
435
436     // clear the previous link state modification
437     for (i=0; i<req->path.top - 1; i++)
438     {
439         int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
440         DBevalLSONRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
441             lsList[lnk], lsList[lnk], req);
442     }
443
444     // clear the old path ...
445     req->path.top = 0;
446
447     // ok we found a path ...
448     printf("Cost = %f, Path = ", dn->val);
449
450     while (dn != NULL && dn->from != NULL)
451     {
452         longListPushBack(&(req->path), dn->node->id);
453         dn = dn->from;
454     }
455     longListPushBack(&(req->path), dn->node->id);
456
457     // revert the path
458     for (i=0; i<(req->path.top + 1)/2; i++)
459     {
460         int tmp = req->path.cont[i];
461         req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
462         req->path.cont[req->path.top - 1 - i] = tmp;
463     }
464
465     for (i=0; i<req->path.top - 1; i++)
466     {
467         int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
468         DBevalLSONSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
469             lsList[lnk], lsList[lnk], req);
470     }

```



```

471
472
473         for (i=0; i<req->path.top - 1; i++)
474         {
475             printf("%ld - ", req->path.cont[i]);
476         }
477         printf("%ld\n", req->path.cont[i]);
478     }
479     else
480     {
481         // oups we have to reject the request ...
482
483     }
484 }
485 }
486 }
487 }
488 else if (type == GLOBAL)
489 {
490
491     // init the list of request to return;
492     lspRequestListResize(reqList, 1); // should not be required !
493     req = lspRequestListGet(reqList, 0);
494
495     req->primID = lsp->id;
496     req->type = GLOBAL_BACK;
497     if (lsp->precedence + 1 < NB_PREEMPTION)
498         req->precedence = lsp->precedence + 1;
499     else
500         req->precedence = lsp->precedence;
501     memmove(&(req->bw), &(lsp->bw), NB_OA * sizeof(double));
502
503     start = lsp->path.cont[0];
504
505     pType = NODE_FAILURE;
506
507     // mark the forbidden links
508     forbiddenLinks.top = 0;
509
510     if (pType == NODE_FAILURE)
511     {
512         // don't remove first and last node !!!
513         for (i=1; i<lsp->path.top-1; i++)
514         {
515             neigh = DBgetNodeInNeighb(dataBase, lsp->path.cont[i]);
516             if (neigh == NULL)
517             {
518                 addError(CRITICAL, "The protected node must have some neighbour in %s at line %d",
519                     __FILE__, __LINE__);
520                 return -1;
521             }
522
523             for (j=0; j<neigh->top; ++j)
524             {
525                 longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[j], lsp->path.
526             }
527         }
528
529         // last link in the path must be removed !!!
530         longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[lsp->path.top-2], 1
531     }
532 }
533 else if (pType == LINK_FAILURE)
534 {
535     for (i=1; i<lsp->path.top; i++)
536     {
537         longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], lsp->path

```

```

538     }
539 }
540
541 // clear the PQ;
542 while (CPpopTop(&toBeTreated) != NULL);
543
544 // clear all the marks
545 for (i=0; i<dataBase->nodeVec.top; i++)
546 {
547     nodeList[i]->marked = FALSE;
548     nodeList[i]->val = -1;
549     nodeList[i]->from = NULL;
550 }
551
552 // push the first node on the PQ
553 CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
554
555 reachPrimary = FALSE;
556 while (reachPrimary == FALSE)
557 {
558     if ((dn = CPpopTop(&toBeTreated)) == NULL)
559     {
560         // Oups ... impossible to reach the primary
561         // if we are in node protection mode, switch back to link protection
562         if (pType == NODE_FAILURE)
563         {
564             printf("Oups ... switching protection ...\n");
565
566             pType = LINK_FAILURE;
567
568             // mark the forbidden links
569             forbiddenLinks.top = 0;
570             for (i=1; i<lsp->path.top; i++)
571             {
572                 longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], 1
573             }
574
575             // clear all marked nodes
576             for (i=0; i<dataBase->nodeVec.top; i++)
577             {
578                 nodeList[i]->marked = FALSE;
579                 nodeList[i]->val = -1;
580                 nodeList[i]->from = NULL;
581             }
582
583             // push the first node on the PQ
584             CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
585
586             // re-enter the loop
587             continue;
588         }
589         else
590         {
591             printf("Oups ... no path found ...\n");
592             break;
593         }
594     }
595
596     // as we don't remove marked node immediatelly we may encounter one now so we skip it
597     if (dn->marked == TRUE)
598         continue;
599
600     // mark the node
601     dn->marked = TRUE;
602
603     // check the stop condition
604     if (lsp->path.cont[lsp->path.top-1] == dn->node->id)

```

```

605         reachPrimary = TRUE;
606
607         // we have finished ... leave the while loop
608         if (reachPrimary == TRUE)
609             break;
610
611         // find the neighbours
612         neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);
613
614         if (neigh != NULL)
615         {
616             for (i=0; i<neigh->top; ++i)
617             {
618                 int id;
619
620                 // check if the node is not already marked
621                 if (nodeList[neigh->cont[i]]->marked == TRUE)
622                     continue;
623
624                 // check if the link is valid
625                 id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
626                 for (j=0; j<forbiddenLinks.top; ++j)
627                     if (forbiddenLinks.cont[j] == id)
628                         break;
629
630                 if (j != forbiddenLinks.top)
631                     continue;
632
633                 // ok now update the node ...
634                 newVal = dn->val + computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn,
635
636                 if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeList[neigh-
637                 {
638                     nodeList[neigh->cont[i]]->val = newVal;
639                     nodeList[neigh->cont[i]]->from = dn;
640                     CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
641                 }
642             }
643         }
644     }
645
646     if (reachPrimary == TRUE)
647     {
648         req = lspRequestListGet(reqList, 0);
649
650         // clear the previous link state modification
651         for (i=0; i<req->path.top - 1; i++)
652         {
653             int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
654             DBevallSONRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
655                             lsList[lnk], lsList[lnk], req);
656         }
657
658         // clear the old path ...
659         req->path.top = 0;
660
661         // ok we found a path ...
662         printf("Cost = %f, Path = ", dn->val);
663
664         while (dn != NULL && dn->from != NULL)
665         {
666             longListPushBack(&(req->path), dn->node->id);
667             dn = dn->from;
668         }
669         longListPushBack(&(req->path), dn->node->id);
670
671         // revert the path

```

```

672         for (i=0; i<(req->path.top + 1)/2; i++)
673         {
674             int tmp = req->path.cont[i];
675             req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
676             req->path.cont[req->path.top - 1 - i] = tmp;
677         }
678
679         for (i=0; i<req->path.top - 1; i++)
680         {
681             int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
682             DBevalLSONSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
683                             lsList[lnk], lsList[lnk], req);
684         }
685
686
687         for (i=0; i<req->path.top - 1; i++)
688         {
689             printf("%ld - ", req->path.cont[i]);
690         }
691         printf("%ld\n", req->path.cont[i]);
692     }
693     else
694     {
695         // oups we have to reject the request ...
696
697
698     }
699 }
700 else if (type == NONE)
701 {
702     addError(INFO, "Oups no backup were requested in %s at line %d",
703              __FILE__, __LINE__);
704 }
705 else
706 {
707     // error
708     addError(WARNING, "Unknown backup type in %s at line %d",
709              __FILE__, __LINE__);
710 }
711
712 #if defined LINUX && defined TIMING && defined TIME3
713 gettimeofday(&t2, &tz);
714 fprintf(stderr, "Time for calculation of backups paths : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000
715          (t2.tv_usec - t1.tv_usec) / 1000.0);
716 #endif
717
718 // Clean Up
719
720 // clear the PrioQueue
721 CPendPQ(&toBeTreated);
722
723 // clear the forbiddenLinks
724 longListEnd(&forbiddenLinks);
725
726 // clear the local copy ...
727 for (i=0; i<dataBase->linkSrcVec.top; i++)
728 {
729     if (lsList[i] != NULL)
730     {
731         if (DBlinkStateDestroy(lsList[i]) < 0)
732         {
733             addError(CRITICAL, "Something went wrong while clearing a structure in %s at line %d",
734                      __FILE__, __LINE__);
735         }
736     }
737 }
738 }
739 }
740

```

```

741     if (lsList != NULL)
742         free(lsList);
743
744     // free the dijkNodes
745     for (i=0; i<dataBase->nodeVec.top; i++)
746     {
747         if (nodeList[i] != NULL)
748         {
749             free(nodeList[i]);
750         }
751     }
752
753     if (nodeList)
754         free(nodeList);
755
756     return 0;
757 }

```

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, DBevalSONSetup(), 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().

```

17 {
18     CPDijkNode* ptr;
19     LSPRequest newReq;
20     DBLinkState newLS;
21     long linkID,i,j;
22     double alpha, beta;
23
24     double inc;
25     double bw_before[NB_OA];
26     double bw_after[NB_OA];
27     double bw_tot_bef=0, bw_tot_aft=0;
28     double bw_tot=0;
29
30     if (DBlinkStateInit(&newLS) < 0)
31     {
32         addError(CRITICAL,"Unable to init a link state in %s at line %d",
33             __FILE__,__LINE__);
34         return -1;
35     }
36
37     beta = 1.0/(1+10);
38     alpha = 1 - beta;
39
40     // bandwidth increment
41     // -----
42
43     lspRequestInit(&newReq);
44     newReq.primID = lsp->id;
45
46     if (lsp->precedence + 1 < NB_PREEMPTION)

```

```

47     newReq.precedence = lsp->precedence + 1;
48 else
49     newReq.precedence = lsp->precedence;
50
51     memmove(&(newReq.bw), &(lsp->bw), NB_OA * sizeof(double));
52
53     if (type == LOCAL)
54         newReq.type = LOCAL_BACK;
55     else if (type == GLOBAL)
56         newReq.type = GLOBAL_BACK;
57     else
58     {
59         addError(WARNING, "Unknown backup type in %s at line %d",
60             __FILE__, __LINE__);
61         return -1;
62     }
63
64     longListPushBack(&(newReq.path), dst);
65
66     ptr = dn;
67     while (ptr != NULL)
68     {
69         longListPushBack(&(newReq.path), ptr->node->id);
70         ptr = ptr->from;
71     }
72
73     // now reverse the path
74     for (i=0; i<(newReq.path.top + 1)/2; i++)
75     {
76         int tmp = newReq.path.cont[i];
77         newReq.path.cont[i] = newReq.path.cont[newReq.path.top - 1 - i];
78         newReq.path.cont[newReq.path.top - 1 - i] = tmp;
79     }
80
81     // eval the impact of the addition
82     if ((linkID = DBgetLinkID(dataBase, src, dst)) >= 0 &&
83         DBevalLSOnSetup(dataBase, src, dst, &newLS, lsList[linkID], &newReq) >= 0)
84     {
85         for (i=0; i<NB_OA; ++i)
86         {
87             bw_tot += lsp->bw[i];
88
89             bw_before[i] = 0;
90             bw_after[i] = 0;
91             for (j=0; j<NB_PREEMPTION; ++j)
92             {
93                 bw_after[i] += newLS.rbw[i][j];
94                 bw_before[i] += lsList[linkID]->rbw[i][j];
95             }
96             bw_tot_bef += bw_before[i];
97             bw_tot_aft += bw_after[i];
98         }
99
100         // printf("%ld-%ld : %f\n", src, dst, bw_after - bw_before);
101     }
102     else
103     {
104         addError(WARNING, "Error while computing new link state in %s at line %d",
105             __FILE__, __LINE__);
106         return -1;
107     }
108 }
109
110 lspRequestEnd(&newReq);
111
112 DBlinkStateEnd(&newLS);
113

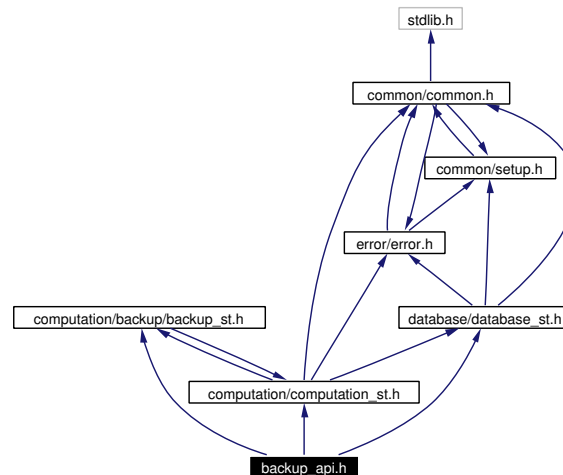
```

```
114     // Capacity constrain ....
115     for (i=0; i<NB_OA; i++)
116         if (bw_after[i] > newLS.cap[i]) {
117             return -1;
118         }
119
120     inc = bw_tot_aft - bw_tot_bef;
121     if (inc < 0) {
122         inc = 0;
123     }
124
125     // ----- Hop count -----
126     // -----
127
128     // if we merge with the primary we have to account for the remaining of the path
129
130     for (i=0; i<lsp->primPath.top; i++) {
131         if (lsp->primPath.cont[i] == dst) {
132             break;
133         }
134     }
135
136     return (alpha * inc) + (beta * (lsp->primPath.top + 1 - i) * bw_tot);
137 }
```

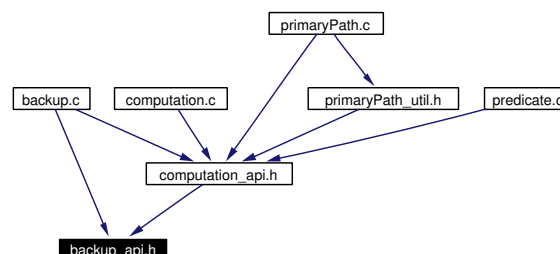
4.4 backup_api.h File Reference

```
#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.

```

147 {
148     long i,j,pNodeIndex,start,pNode;
149     DBLinkState *ls, *oldLS;
150     DBLinkState** lsList;
151     CPDijkNode *dn=NULL;
152     CPDijkNode** nodeList;
153     bool reachPrimary;
154     CPPrioQueue toBeTreated;
155     LongList* neigh;
156     LongList forbiddenLinks;
157     DBLabelSwitchedPath* lsp;
158     LSPRequest* req=NULL;
159     double newVal;
160     int pType=0;
161     int src, dst;
162
163     enum {NODE_FAILURE, LINK_FAILURE};
164
165 #if defined LINUX && defined TIMING && defined TIME3
166     struct timezone tz;
167     struct timeval t1,t2;
168 #endif
169
170     if (dataBase == NULL || reqList == NULL)
171     {
172         addError(CRITICAL,"Wrong argument in %s at line %d",
173             __FILE__,__LINE__);
174         return -1;
175     }
176
177     if (lspID < 0 || ((lsp = DBgetLSP(dataBase, lspID)) == NULL))
178     {
179         addError(CRITICAL,"Cannot find lsp ID in %s at line %d",
180             __FILE__,__LINE__);
181         return -1;
182     }
183
184     if ((lsList = calloc(dataBase->linkSrcVec.top, sizeof(DBLinkState*))) == NULL)
185     {
186         addError(CRITICAL,"Cannot allocate a required structure in %s at line %d",
187             __FILE__,__LINE__);

```

```

188     return -1;
189 }
190
191 if ((nodeList = calloc(dataBase->nodeVec.top, sizeof(CPDijkNode*))) == NULL)
192 {
193     addError(CRITICAL, "Cannot allocate a required structure in %s at line %d",
194             __FILE__, __LINE__);
195     free (lsList);
196     return -1;
197 }
198
199 // duplicate all the link-states
200 for (i=0; i<dataBase->linkSrcVec.top; i++)
201 {
202     src = DBgetLinkSrc(dataBase, i);
203     dst = DBgetLinkDst(dataBase, i);
204
205     if (src != -1 && dst != -1)
206     {
207         if ((oldLS = DBgetLinkState(dataBase, src, dst)) == NULL)
208         {
209             addError(WARNING, "Oups there should be a link-state here in %s at line %d",
210                     __FILE__, __LINE__);
211             continue;
212         }
213
214         if ((ls = DBlinkStateNew()) == NULL)
215         {
216             addError(CRITICAL, "Cannot duplicate all the link-states in %s at line %d",
217                     __FILE__, __LINE__);
218             continue;
219         }
220
221         if (DBlinkStateCopy(ls, oldLS) < 0)
222         {
223             addError(CRITICAL, "Something went wrong while copying in %s at line %d",
224                     __FILE__, __LINE__);
225             DBlinkStateDestroy(ls);
226             continue;
227         }
228
229         lsList[i] = ls;
230     }
231     else
232     {
233         addError(INFO, "Warning there is no link numbered %ld : src = %ld, dst = %ld .... in %s at
234             __FILE__, __LINE__);
235     }
236 }
237
238 // create the Dijk Nodes ... used for the computation
239 for (i=0; i<dataBase->nodeVec.top; i++)
240 {
241     if (dataBase->nodeVec.cont[i] != NULL)
242     {
243         if ((dn = calloc(1, sizeof(CPDijkNode))) == NULL)
244         {
245             addError(CRITICAL, "Cannot create the Dijk nodes in %s at line %d",
246                     __FILE__, __LINE__);
247             continue;
248         }
249
250         dn->node = dataBase->nodeVec.cont[i];
251         dn->val = -1;
252         dn->marked = FALSE;
253
254         nodeList[i] = dn;

```

```

255     }
256 }
257
258 printf("Primary path :");
259 for (i=0; i<lsp->path.top - 1; ++i)
260 {
261     printf("%ld - ", lsp->path.cont[i]);
262 }
263 printf("%ld\n", lsp->path.cont[i]);
264
265 // now start the calculation ...
266
267 #if defined LINUX && defined TIMING && defined TIME3
268     gettimeofday(&t1, &tz);
269 #endif
270
271 // init a PrioQueue
272 CPinitPQ(&toBeTreated);
273
274 // init the forbiddenLinks;
275 longListInit(&forbiddenLinks, -1);
276
277 if (type == LOCAL)
278 {
279     // init the list of request to return;
280     lspRequestListResize(reqList, lsp->path.top-1);
281     for (i=0; i<lsp->path.top-1; ++i)
282     {
283         req = lspRequestListGet(reqList, i);
284
285         req->primID = lsp->id;
286         req->type = LOCAL_BACK;
287         if (lsp->precedence + 1 < NB_PREEMPTION)
288             req->precedence = lsp->precedence + 1;
289         else
290             req->precedence = lsp->precedence;
291         memmove(&(req->bw), &(lsp->bw), NB_OA * sizeof(double));
292     }
293
294     // for (pNodeIndex=1; pNodeIndex<req->path.top; ++pNodeIndex) // start at 1 because we cannot
295     for (pNodeIndex=lsp->path.top - 1; pNodeIndex>0; --pNodeIndex)
296     {
297         pNode = lsp->path.cont[pNodeIndex];
298         start = lsp->path.cont[pNodeIndex - 1];
299
300         pType = NODE_FAILURE;
301
302         // mark the forbidden links
303         forbiddenLinks.top = 0;
304
305         if (pType == NODE_FAILURE)
306         {
307             neigh = DBgetNodeInNeighb(dataBase, pNode);
308             if (neigh == NULL)
309             {
310                 addError(CRITICAL, "The protected node must have some neighbour in %s at line %d",
311                     __FILE__, __LINE__);
312                 return -1;
313             }
314
315             for (i=0; i<neigh->top; ++i)
316             {
317                 longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[i], pNode));
318             }
319         }
320         else if (pType == LINK_FAILURE)
321         {

```

```

322         longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));
323     }
324
325     // clear the PQ;
326     while (CPpopTop(&toBeTreated) != NULL);
327
328     // clear all the marks
329     for (i=0; i<dataBase->nodeVec.top; i++)
330     {
331         nodeList[i]->marked = FALSE;
332         nodeList[i]->val = -1;
333         nodeList[i]->from = NULL;
334     }
335
336     // push the first node on the PQ
337     CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
338
339     reachPrimary = FALSE;
340     while (reachPrimary == FALSE)
341     {
342         if ((dn = CPpopTop(&toBeTreated)) == NULL)
343         {
344             // Oups ... impossible to reach the primary
345             // if we are in node protection mode, switch back to link protection
346             if (pType == NODE_FAILURE)
347             {
348                 pType = LINK_FAILURE;
349
350                 // mark the forbidden links
351                 forbiddenLinks.top = 0;
352                 longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, start, pNode));
353
354                 // clear all marked nodes
355                 for (i=0; i<dataBase->nodeVec.top; i++)
356                 {
357                     nodeList[i]->marked = FALSE;
358                     nodeList[i]->val = -1;
359                     nodeList[i]->from = NULL;
360                 }
361
362                 // push the first node on the PQ
363                 CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
364
365                 // re-enter the loop
366                 continue;
367             }
368             else
369             {
370                 break;
371             }
372         }
373
374         // as we don't remove marked node immediatelly we may encounter one now so we skip it
375         if (dn->marked == TRUE)
376             continue;
377
378         // mark the node
379         dn->marked = TRUE;
380
381         // check the stop condition
382         for (i=pNodeIndex; i<lsp->path.top; ++i)
383             if (lsp->path.cont[i] == dn->node->id)
384             {
385                 reachPrimary = TRUE;
386                 break;
387             }
388     }

```

```

389         // we have finished ... leave the while loop
390         if (reachPrimary == TRUE)
391             break;
392
393         // find the neighbours
394         neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);
395
396         if (neigh != NULL)
397         {
398             for (i=0; i<neigh->top; ++i)
399             {
400                 int id;
401                 double cost;
402
403                 // check if the node is not already marked
404                 if (nodeList[neigh->cont[i]]->marked == TRUE)
405                     continue;
406
407                 // check if the link is valid
408                 id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
409                 for (j=0; j<forbiddenLinks.top; ++j)
410                     if (forbiddenLinks.cont[j] == id)
411                         break;
412
413                 if (j != forbiddenLinks.top)
414                     continue;
415
416                 // ok now update the node ...
417                 cost = computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn, lsp, ty);
418                 if (cost >= 0) {
419                     newVal = dn->val + cost;
420
421                     if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeList[neigh->cont[i]]->val))
422                     {
423                         nodeList[neigh->cont[i]]->val = newVal;
424                         nodeList[neigh->cont[i]]->from = dn;
425                         CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
426                     }
427                 }
428             }
429         }
430     }
431
432     if (reachPrimary == TRUE)
433     {
434         req = lspRequestListGet(reqList, pNodeIndex-1);
435
436         // clear the previous link state modification
437         for (i=0; i<req->path.top - 1; i++)
438         {
439             int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
440             DBevalLSOnRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
441                             lsList[lnk], lsList[lnk], req);
442         }
443
444         // clear the old path ...
445         req->path.top = 0;
446
447         // ok we found a path ...
448         printf("Cost = %f, Path = ", dn->val);
449
450         while (dn != NULL && dn->from != NULL)
451         {
452             longListPushBack(&(req->path), dn->node->id);
453             dn = dn->from;
454         }
455         longListPushBack(&(req->path), dn->node->id);

```

```

456
457         // revert the path
458         for (i=0; i<(req->path.top + 1)/2; i++)
459         {
460             int tmp = req->path.cont[i];
461             req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
462             req->path.cont[req->path.top - 1 - i] = tmp;
463         }
464
465         for (i=0; i<req->path.top - 1; i++)
466         {
467             int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
468             DBevalSONSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
469                             lsList[lnk], lsList[lnk], req);
470         }
471
472
473         for (i=0; i<req->path.top - 1; i++)
474         {
475             printf("%ld - ", req->path.cont[i]);
476         }
477         printf("%ld\n", req->path.cont[i]);
478     }
479     else
480     {
481         // ouns we have to reject the request ...
482
483     }
484 }
485
486 }
487 }
488 else if (type == GLOBAL)
489 {
490
491     // init the list of request to return;
492     lspRequestListResize(reqList, 1); // should not be required !
493     req = lspRequestListGet(reqList, 0);
494
495     req->primID = lsp->id;
496     req->type = GLOBAL_BACK;
497     if (lsp->precedence + 1 < NB_PREEMPTION)
498         req->precedence = lsp->precedence + 1;
499     else
500         req->precedence = lsp->precedence;
501     memmove(&(req->bw), &(lsp->bw), NB_OA * sizeof(double));
502
503     start = lsp->path.cont[0];
504
505     pType = NODE_FAILURE;
506
507     // mark the forbidden links
508     forbiddenLinks.top = 0;
509
510     if (pType == NODE_FAILURE)
511     {
512         // don't remove first and last node !!!
513         for (i=1; i<lsp->path.top-1; i++)
514         {
515             neigh = DBgetNodeInNeighb(dataBase, lsp->path.cont[i]);
516             if (neigh == NULL)
517             {
518                 addError(CRITICAL, "The protected node must have some neighbour in %s at line %d",
519                         __FILE__, __LINE__);
520                 return -1;
521             }
522         }

```

```

523         for (j=0; j<neigh->top; ++j)
524         {
525             longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, neigh->cont[j], lsp->path.
526         }
527     }
528
529     // last link in the path must be removed !!!
530     longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[lsp->path.top-2], 1
531
532 }
533 else if (pType == LINK_FAILURE)
534 {
535     for (i=1; i<lsp->path.top; i++)
536     {
537         longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], lsp->path
538     }
539 }
540
541 // clear the PQ;
542 while (CPpopTop(&toBeTreated) != NULL);
543
544 // clear all the marks
545 for (i=0; i<dataBase->nodeVec.top; i++)
546 {
547     nodeList[i]->marked = FALSE;
548     nodeList[i]->val = -1;
549     nodeList[i]->from = NULL;
550 }
551
552 // push the first node on the PQ
553 CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
554
555 reachPrimary = FALSE;
556 while (reachPrimary == FALSE)
557 {
558     if ((dn = CPpopTop(&toBeTreated)) == NULL)
559     {
560         // Oups ... impossible to reach the primary
561         // if we are in node protection mode, switch back to link protection
562         if (pType == NODE_FAILURE)
563         {
564             printf("Oups ... switching protection ...\n");
565
566             pType = LINK_FAILURE;
567
568             // mark the forbidden links
569             forbiddenLinks.top = 0;
570             for (i=1; i<lsp->path.top; i++)
571             {
572                 longListPushBack(&forbiddenLinks, DBgetLinkID(dataBase, lsp->path.cont[i-1], 1
573             }
574
575             // clear all marked nodes
576             for (i=0; i<dataBase->nodeVec.top; i++)
577             {
578                 nodeList[i]->marked = FALSE;
579                 nodeList[i]->val = -1;
580                 nodeList[i]->from = NULL;
581             }
582
583             // push the first node on the PQ
584             CPinsertPQ(&toBeTreated, nodeList[start], (nodeList[start]->val=0));
585
586             // re-enter the loop
587             continue;
588         }
589     }

```

```

590         {
591             printf("Oops ... no path found ...\n");
592             break;
593         }
594     }
595
596     // as we don't remove marked node immediately we may encounter one now so we skip it
597     if (dn->marked == TRUE)
598         continue;
599
600     // mark the node
601     dn->marked = TRUE;
602
603     // check the stop condition
604     if (lsp->path.cont[lsp->path.top-1] == dn->node->id)
605         reachPrimary = TRUE;
606
607     // we have finished ... leave the while loop
608     if (reachPrimary == TRUE)
609         break;
610
611     // find the neighbours
612     neigh = DBgetNodeOutNeighb(dataBase, dn->node->id);
613
614     if (neigh != NULL)
615     {
616         for (i=0; i<neigh->top; ++i)
617         {
618             int id;
619
620             // check if the node is not already marked
621             if (nodeList[neigh->cont[i]]->marked == TRUE)
622                 continue;
623
624             // check if the link is valid
625             id = DBgetLinkID(dataBase, dn->node->id, neigh->cont[i]);
626             for (j=0; j<forbiddenLinks.top; ++j)
627                 if (forbiddenLinks.cont[j] == id)
628                     break;
629
630             if (j != forbiddenLinks.top)
631                 continue;
632
633             // ok now update the node ...
634             newVal = dn->val + computeCost(dataBase, lsList, dn->node->id, neigh->cont[i], dn,
635
636             if (nodeList[neigh->cont[i]]->val == -1 || (newVal > 0 && newVal < nodeList[neigh->
637             {
638                 nodeList[neigh->cont[i]]->val = newVal;
639                 nodeList[neigh->cont[i]]->from = dn;
640                 CPinsertPQ(&toBeTreated, nodeList[neigh->cont[i]], newVal);
641             }
642         }
643     }
644 }
645
646 if (reachPrimary == TRUE)
647 {
648     req = lspRequestListGet(reqList, 0);
649
650     // clear the previous link state modification
651     for (i=0; i<req->path.top - 1; i++)
652     {
653         int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
654         DBevallSONRemove(dataBase, req->path.cont[i], req->path.cont[i+1],
655                         lsList[lnk], lsList[lnk], req);
656     }

```



```

657
658         // clear the old path ...
659         req->path.top = 0;
660
661         // ok we found a path ...
662         printf("Cost = %f, Path = ", dn->val);
663
664         while (dn != NULL && dn->from != NULL)
665         {
666             longListPushBack(&(req->path), dn->node->id);
667             dn = dn->from;
668         }
669         longListPushBack(&(req->path), dn->node->id);
670
671         // revert the path
672         for (i=0; i<(req->path.top + 1)/2; i++)
673         {
674             int tmp = req->path.cont[i];
675             req->path.cont[i] = req->path.cont[req->path.top - 1 - i];
676             req->path.cont[req->path.top - 1 - i] = tmp;
677         }
678
679         for (i=0; i<req->path.top - 1; i++)
680         {
681             int lnk = DBgetLinkID(dataBase, req->path.cont[i], req->path.cont[i+1]);
682             DBevalLSONSetup(dataBase, req->path.cont[i], req->path.cont[i+1],
683                             lsList[lnk], lsList[lnk], req);
684         }
685
686
687         for (i=0; i<req->path.top - 1; i++)
688         {
689             printf("%ld - ", req->path.cont[i]);
690         }
691         printf("%ld\n", req->path.cont[i]);
692     }
693     else
694     {
695         // ouns we have to reject the request ...
696
697     }
698 }
699 }
700 else if (type == NONE)
701 {
702     addError(INFO, "Oups no backup were requested in %s at line %d",
703              __FILE__, __LINE__);
704 }
705 else
706 {
707     // error
708     addError(WARNING, "Unknown backup type in %s at line %d",
709              __FILE__, __LINE__);
710 }
711
712 #if defined LINUX && defined TIMING && defined TIME3
713     gettimeofday(&t2, &tz);
714     fprintf(stderr, "Time for calculation of backups paths : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000
715             (t2.tv_usec - t1.tv_usec) / 1000.0);
716 #endif
717
718 // Clean Up
719
720 // clear the PrioQueue
721 CPendPQ(&toBeTreated);
722
723 // clear the forbiddenLinks

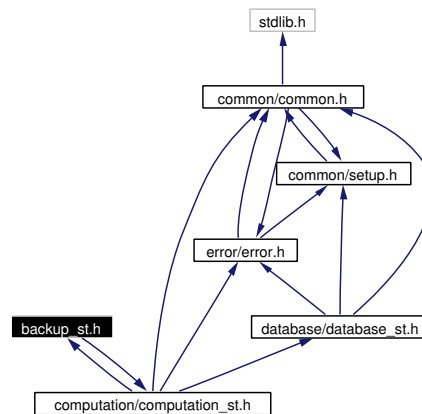
```

```
726     longListEnd(&forbiddenLinks);
727
728     // clear the local copy ...
729     for (i=0; i<dataBase->linkSrcVec.top; i++)
730     {
731         if (lsList[i] != NULL)
732         {
733             if (DBlinkStateDestroy(lsList[i]) < 0)
734             {
735                 addError(CRITICAL, "Something went wrong while clearing a structure in %s at line %d",
736                     __FILE__, __LINE__);
737             }
738         }
739     }
740
741     if (lsList != NULL)
742         free(lsList);
743
744     // free the dijkNodes
745     for (i=0; i<dataBase->nodeVec.top; i++)
746     {
747         if (nodeList[i] != NULL)
748         {
749             free(nodeList[i]);
750         }
751     }
752
753     if (nodeList)
754         free(nodeList);
755
756     return 0;
757 }
```

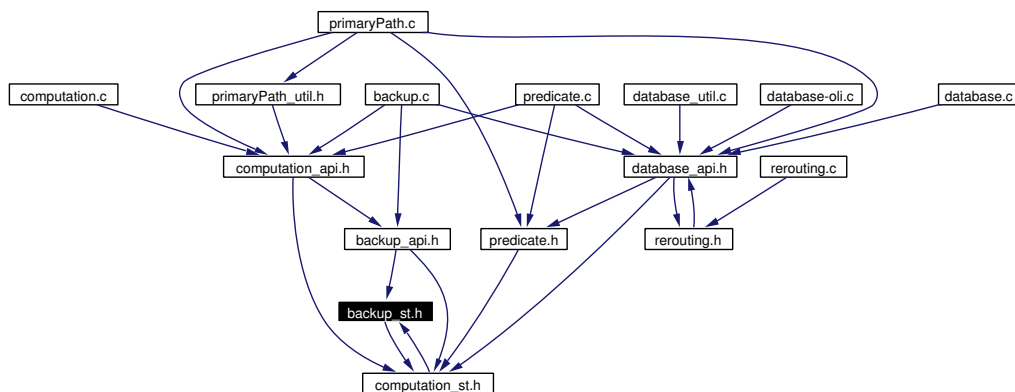
4.5 backup_st.h File Reference

```
#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

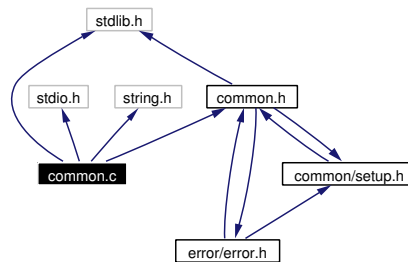
Definition at line 6 of file backup_st.h.

```
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()`.

```

530 {
531     double *ptr=NULL;
532
533     if (dst == NULL || dst->cont == NULL ||
534         src == NULL || src->cont == NULL)
535     {
536         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
537             __FILE__,__LINE__);
538         return -1;
539     }
540
541     if (dst->size < src->size)
542     {
543         if ((ptr=realloc(dst->cont,src->size*sizeof(double)))==NULL)
544         {
545             addError(CRITICAL,"Critical lack of memory in %s at line %d",
546                 __FILE__,__LINE__);
547             return -1;
548         }
549         else
550         {
551             dst->cont=ptr;
552             dst->size=src->size;
553         }
554     }
555
556     memcpy(dst->cont,src->cont,src->size*sizeof(double));
557     dst->top=src->top;
558
559     return 0;
560 }
```

4.6.1.2 `int dblVecDestroy (DoubleVec * vec)`

Definition at line 514 of file common.c.

References `addError()`, `DoubleVec::cont`, `CRITICAL`, and `free`.

```

515 {
516     if (vec == NULL || vec->cont == NULL)
517     {
518         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
519             __FILE__,__LINE__);
520         return -1;
521     }
522
523     free(vec->cont);
```

```
524     free(vec);
525
526     return 0;
527 }
```

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()`.

```
498 {
499     if (vec == NULL || vec->cont == NULL)
500     {
501         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
502                 __FILE__, __LINE__);
503         return -1;
504     }
505
506     free(vec->cont);
507     vec->cont = NULL;
508     vec->size = 0;
509     vec->top = 0;
510
511     return 0;
512 }
```

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`.

```
641 {
642     if (vec == NULL || vec->cont == NULL || val == NULL)
643     {
644         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
645                 __FILE__, __LINE__);
646         return -1;
647     }
648
649     if (index < 0)
650     {
651         addError(CRITICAL, "Bad argument (index < 0) in %s at line %d",
652                 __FILE__, __LINE__);
653         return -1;
654     }
655
656     if (index >= vec->size)
657     {
658         addError(CRITICAL, "Bad argument (wrong index) in %s at line %d",
659                 __FILE__, __LINE__);
660         return -1;
661     }
662
663     *val = vec->cont[index];
664
665     return 0;
666 }
667 }
```

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().

```

470 {
471     void* ptr=NULL;
472
473     if (vec == NULL)
474     {
475         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
476                 __FILE__,__LINE__);
477         return -1;
478     }
479
480     if (size == -1)
481         size = DBLVEC_INITSIZE;
482
483     if ((ptr = calloc(size,sizeof(double))) == NULL)
484     {
485         addError(CRITICAL,"Critical lack of memory in %s at line %d",
486                 __FILE__,__LINE__);
487         return -1;
488     }
489
490     vec->size = size;
491     vec->top = 0;
492     vec->cont = ptr;
493
494     return 0;
495 }
```

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.

```

440 {
441     DoubleVec* vec=NULL;
442     void* ptr=NULL;
443
444     if ((vec = calloc(1,sizeof(DoubleVec))) == NULL)
445     {
446         addError(CRITICAL,"Critical lack of memory in %s at line %d",
447                 __FILE__,__LINE__);
448         return NULL;
449     }
450
451     if (size == -1)
452         size = DBLVEC_INITSIZE;
453
454     if ((ptr = calloc(size,sizeof(double))) == NULL)
455     {
456         addError(CRITICAL,"Critical lack of memory in %s at line %d",
457                 __FILE__,__LINE__);
458         free(vec);
459         return NULL;
460     }
```



```
460     }
461
462     vec->size = size;
463     vec->top = 0;
464     vec->cont = ptr;
465
466     return vec;
467 }
```

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`.

```
593 {
594     if (vec == NULL || vec->cont == NULL || val == NULL)
595     {
596         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
597                 __FILE__, __LINE__);
598         return -1;
599     }
600
601     if (vec->top == 0)
602     {
603         addError(CRITICAL, "Pop on empty stack in %s at line %d",
604                 __FILE__, __LINE__);
605         return -1;
606     }
607
608     *val = vec->cont[vec->top - 1];
609     vec->top--;
610
611     return 0;
612 }
```

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`.

```
563 {
564     void* ptr=NULL;
565
566     if (vec == NULL || vec->cont == NULL)
567     {
568         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
569                 __FILE__, __LINE__);
570         return -1;
571     }
572
573     if (vec->top >= vec->size)
574     {
575         if ((ptr = realloc(vec->cont, vec->size *
576                           2 * sizeof(double))) == NULL)
577         {
578             addError(CRITICAL, "Critical lack of memory in %s at line %d",
579                     __FILE__, __LINE__);
580             return -1;
581         }
582     }
```

```

582
583     vec->size *= 2;
584     vec->cont = ptr;
585 }
586
587 vec->cont[vec->top++] = val;
588
589 return 0;
590 }

```

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().

```

616 {
617     void* ptr=NULL;
618
619     if (vec == NULL || vec->cont == NULL)
620     {
621         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
622             __FILE__,__LINE__);
623         return -1;
624     }
625
626     if ((ptr = realloc(vec->cont, newsize*sizeof(double))) == NULL)
627     {
628         addError(CRITICAL,"Critical lack of memory in %s at line %d",
629             __FILE__,__LINE__);
630         return -1;
631     }
632
633     vec->cont = ptr;
634     memset(ptr+ (vec->size * sizeof(double)), 0, (newsize - vec->size)*sizeof(double));
635     vec->size = newsize;
636
637     return 0;
638 }

```

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.

```

670 {
671     if (vec == NULL || vec->cont == NULL)
672     {
673         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
674             __FILE__,__LINE__);
675         return -1;
676     }
677
678     if (index < 0)
679     {
680         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
681             __FILE__,__LINE__);

```

```

682     return -1;
683 }
684
685 if (index >= vec->size)
686 {
687     if (dblVecResize(vec,max(vec->size * 2,index+1))<0)
688     {
689         addError(CRITICAL,"Unable to resize double vector in %s at line %d",
690             __FILE__,__LINE__);
691         return -1;
692     }
693 }
694
695 vec->cont[index] = val;
696 vec->top = max(vec->top, index+1);
697
698 return 0;
699 }

```

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().

```

348 {
349     if ((* (long*)a)<(* (long*)b))
350         return -1;
351     else if ((* (long*)a)>(* (long*)b))
352         return 1;
353     else
354         return 0;
355 }

```

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.

```

282 {
283     if (list == NULL || list->cont == NULL)
284     {
285         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
286             __FILE__,__LINE__);
287         return -1;
288     }
289
290     if (index < 0)
291     {
292         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
293             __FILE__,__LINE__);
294         return -1;
295     }
296
297     if ((list->top >= list->size) || (index >= list->size))
298     {
299         if (longListResize(list,max(list->size * 2,index+1))<0)
300         {
301             addError(CRITICAL,"Unable to resize long vector in %s at line %d",

```

```

302         __FILE__, __LINE__);
303     return -1;
304 }
305 }
306
307 if (index < list->top)
308 {
309     memmove(list->cont+index+1, list->cont+index, (list->top-index) * sizeof(long));
310 }
311 list->cont[index]=val;
312 list->top++;
313
314 return 0;
315 }

```

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().

```

358 {
359     int i=0,j=0;
360     LongList tmpList;
361
362     if (la == NULL || la->cont == NULL ||
363         lb == NULL || lb->cont == NULL ||
364         dest == NULL || dest->cont == NULL)
365     {
366         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
367             __FILE__, __LINE__);
368         return -1;
369     }
370
371     qsort(la->cont, la->top, sizeof(long), &longCompare);
372     qsort(lb->cont, lb->top, sizeof(long), &longCompare);
373
374     if (longListInit(&tmpList, la->top+lb->top) < 0)
375     {
376         addError(CRITICAL, "Unable to initialize temporary list of longs in %s at line %d",
377             __FILE__, __LINE__);
378         return -1;
379     }
380
381     while ((i < la->top) || (j < lb->top))
382     {
383         if ((j >= lb->top) || (i < la->top && la->cont[i] < lb->cont[j]))
384         {
385             longListPushBack(&tmpList, la->cont[i]);
386             i++;
387         }
388         else if ((i >= la->top) || (j < lb->top && la->cont[i] > lb->cont[j]))
389         {
390             longListPushBack(&tmpList, lb->cont[j]);
391             j++;
392         }
393         else if (la->cont[i] == lb->cont[j])
394         {
395             longListPushBack(&tmpList, la->cont[i]);
396             i++;
397             j++;

```

```

398     }
399     else
400     {
401         addError(CRITICAL,"Internal error in %s at line %d",
402             __FILE__,__LINE__);
403         longListEnd(&tmpList);
404         return -1;
405     }
406 }
407
408 if (longListCopy(dest,&tmpList)<0)
409 {
410     addError(CRITICAL,"Unable to create resulting merged list of longs in %s at line %d",
411         __FILE__,__LINE__);
412     longListEnd(&tmpList);
413     return -1;
414 }
415
416 longListEnd(&tmpList);
417
418 return 0;
419 }

```

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\(\)](#).

```

318 {
319     if (list == NULL || list->cont == NULL)
320     {
321         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
322             __FILE__,__LINE__);
323         return -1;
324     }
325
326     if (index < 0)
327     {
328         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
329             __FILE__,__LINE__);
330         return -1;
331     }
332
333     if (index>=list->top)
334     {
335         addError(WARNING,"Removing inexistent list element in %s at line %d",
336             __FILE__,__LINE__);
337         return -1;
338     }
339     else
340     {
341         memmove(list->cont+index,list->cont+index+1,(list->top-index-1) * sizeof(long));
342         list->top--;
343         return 0;
344     }
345 }

```

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().

```

421         {
422
423     if (list == NULL || list->cont == NULL)
424     {
425         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
426             __FILE__, __LINE__);
427         return -1;
428     }
429
430     qsort(list->cont, list->top, sizeof(long), &longCompare);
431
432     return 0;
433 }
```

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.

```

107 {
108     long *ptr=NULL;
109
110     if (dst == NULL || dst->cont == NULL ||
111         src == NULL || src->cont == NULL)
112     {
113         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
114             __FILE__, __LINE__);
115         return -1;
116     }
117
118     if (dst->size < src->size)
119     {
120         if ((ptr=(long*) realloc(dst->cont, src->size*sizeof(long)))==NULL)
121         {
122             addError(CRITICAL, "Critical lack of memory in %s at line %d",
123                 __FILE__, __LINE__);
124             return -1;
125         }
126         else
127         {
128             dst->cont=ptr;
129             dst->size=src->size;
130         }
131     }
132
133     memcpy(dst->cont, src->cont, src->size*sizeof(long));
134     dst->top=src->top;
135
136     return 0;
137 }
```

4.6.1.17 int longVecDestroy (LongVec * vec)

Definition at line 91 of file common.c.

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

```

92 {
93     if (vec == NULL || vec->cont == NULL)
94     {
95         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
96                 __FILE__, __LINE__);
97         return -1;
98     }
99
100     free(vec->cont);
101     free(vec);
102
103     return 0;
104 }

```

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\(\)](#).

```

75 {
76     if (vec == NULL || vec->cont == NULL)
77     {
78         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
79                 __FILE__, __LINE__);
80         return -1;
81     }
82
83     free(vec->cont);
84     vec->cont = NULL;
85     vec->size = 0;
86     vec->top = 0;
87
88     return 0;
89 }

```

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\(\)](#).

```

217 {
218     if (vec == NULL || vec->cont == NULL || val == NULL)
219     {
220         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
221                 __FILE__, __LINE__);
222         return -1;
223     }
224
225     if (index < 0)
226     {
227         addError(CRITICAL, "Bad argument (index < 0) in %s at line %d",
228                 __FILE__, __LINE__);
229         return -1;
230     }
231

```

```

232     if (index >= vec->size)
233     {
234         addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
235             __FILE__,__LINE__);
236         return -1;
237     }
238
239     *val = vec->cont[index];
240
241     return 0;
242
243 }

```

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().

```

47 {
48     void* ptr=NULL;
49
50     if (vec == NULL)
51     {
52         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
53             __FILE__,__LINE__);
54         return -1;
55     }
56
57     if (size == -1)
58         size = LONGVEC_INITSIZE;
59
60     if ((ptr = calloc(size,sizeof(long))) == NULL)
61     {
62         addError(CRITICAL,"Critical lack of memory in %s at line %d",
63             __FILE__,__LINE__);
64         return -1;
65     }
66
67     vec->size = size;
68     vec->top = 0;
69     vec->cont = ptr;
70
71     return 0;
72 }

```

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.

```

17 {
18     LongVec* vec=NULL;
19     void* ptr=NULL;
20
21     if ((vec = calloc(1,sizeof(LongVec))) == NULL)

```



```

22     {
23         addError(CRITICAL,"Critical lack of memory in %s at line %d",
24                 __FILE__,__LINE__);
25         return NULL;
26     }
27
28     if (size == -1)
29         size = LONGVEC_INITSIZE;
30
31     if ((ptr = calloc(size,sizeof(long))) == NULL)
32     {
33         addError(CRITICAL,"Critical lack of memory in %s at line %d",
34                 __FILE__,__LINE__);
35         free(vec);
36         return NULL;
37     }
38
39     vec->size = size;
40     vec->top = 0;
41     vec->cont = ptr;
42
43     return vec;
44 }

```

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](#).

```

170 {
171     if (vec == NULL || vec->cont == NULL || val == NULL)
172     {
173         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
174                 __FILE__,__LINE__);
175         return -1;
176     }
177
178     if (vec->top == 0)
179     {
180         addError(CRITICAL,"Pop on empty stack in %s at line %d",
181                 __FILE__,__LINE__);
182         return -1;
183     }
184
185     *val = vec->cont[vec->top - 1];
186     vec->top--;
187
188     return 0;
189 }

```

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](#).

```

140 {
141     void* ptr=NULL;
142
143     if (vec == NULL || vec->cont == NULL)

```

```

144     {
145         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
146                 __FILE__,__LINE__);
147         return -1;
148     }
149
150     if (vec->top >= vec->size)
151     {
152         if ((ptr = realloc(vec->cont, vec->size *
153                           2 * sizeof(long))) == NULL)
154         {
155             addError(CRITICAL,"Critical lack of memory in %s at line %d",
156                     __FILE__,__LINE__);
157             return -1;
158         }
159
160         vec->size *= 2;
161         vec->cont = ptr;
162     }
163
164     vec->cont[vec->top++] = val;
165
166     return 0;
167 }

```

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().

```

192 {
193     void* ptr=NULL;
194
195     if (vec == NULL || vec->cont == NULL)
196     {
197         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
198                 __FILE__,__LINE__);
199         return -1;
200     }
201
202     if ((ptr = realloc(vec->cont, newsize*sizeof(long))) == NULL)
203     {
204         addError(CRITICAL,"Critical lack of memory in %s at line %d",
205                 __FILE__,__LINE__);
206         return -1;
207     }
208
209     vec->cont = ptr;
210     memset(ptr+ (vec->size * sizeof(long)), 0, (newsize - vec->size)*sizeof(long));
211     vec->size = newsize;
212
213     return 0;
214 }

```

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 {

```

```

743     if (*((long*)(ptr-sizeof(long))) == -1)
744         fprintf(stderr, "Warning already freed\n");
745
746     allocatedMemory-=*((long*)(ptr-sizeof(long)));
747     (*((long*)(ptr-sizeof(long)))=-1;
748     free(ptr-sizeof(long));
749 }

```

4.6.1.28 void* mymalloc (size_t sz)

Definition at line 715 of file common.c.

References allocatedMemory, and malloc.

Referenced by mycalloc().

```

716 {
717     void *ptr;
718
719     if ((ptr=malloc(sz+sizeof(long)))==NULL)
720         return NULL;
721
722     allocatedMemory+=sz;
723     *((long*)ptr)=sz;
724
725     return (ptr+sizeof(long));
726 }

```

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

Definition at line 728 of file common.c.

References allocatedMemory, and realloc.

```

729 {
730     void *retptr;
731
732     if ((retptr=realloc(ptr-sizeof(long),sz+sizeof(long)))==NULL)
733         return NULL;
734
735     allocatedMemory+=sz-*((long*)(retptr));
736     *((long*)retptr)=sz;
737
738     return (retptr+sizeof(long));
739 }

```

4.6.2 Variable Documentation

4.6.2.1 long allocatedMemory = 0

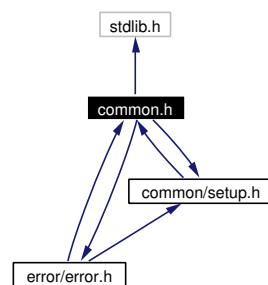
Definition at line 713 of file common.c.

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

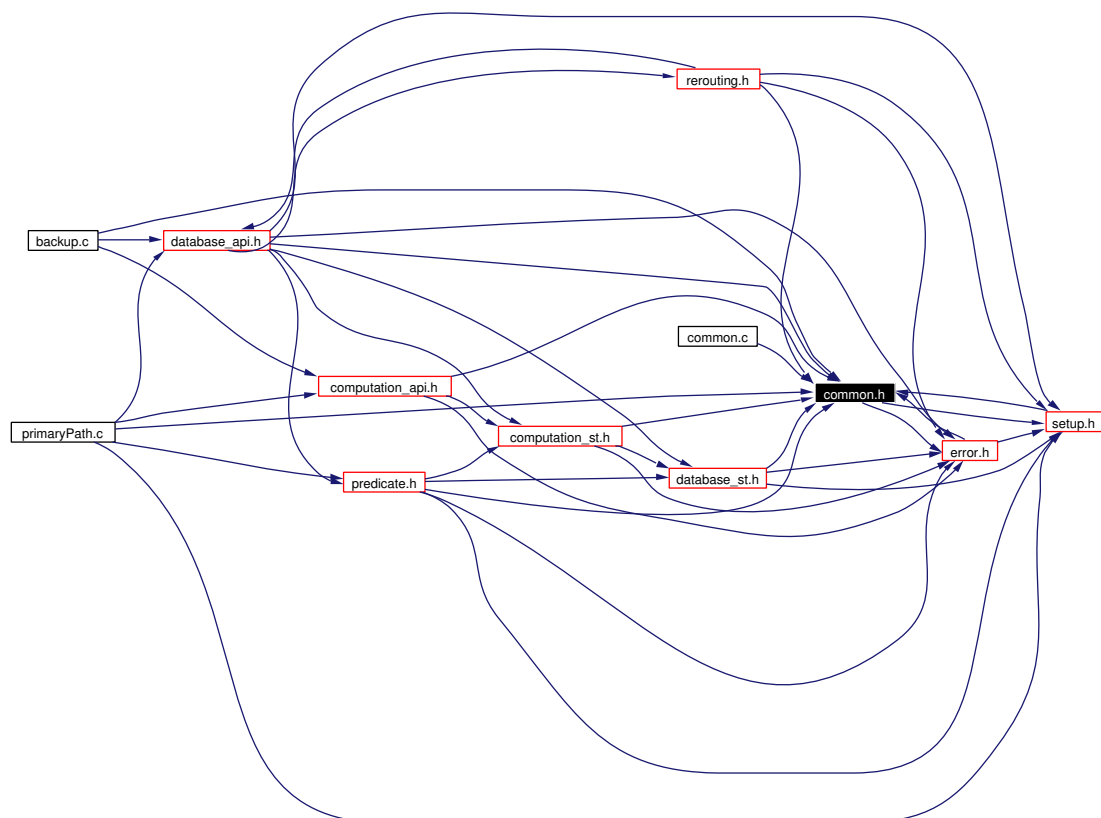
4.7 common.h File Reference

```
#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 lspRequestCopy().

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(), DBlspInit(), DBlspNew(), DBnodeDestroy(), DBnodeEnd(), DBnodeInit(), DBnodeNew(), fillTopo(), isValidRequestLink(), longListMerge(), lspRequestDestroy(), lspRequestEnd(), lspRequestInit(), and lspRequestNew().

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(), DBnodeInit(), DBnodeNew(), fillTopo(), isValidRequestLink(), longListMerge(), lspRequestInit(), and lspRequestNew().

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().

```
530 {
531     double *ptr=NULL;
532
533     if (dst == NULL || dst->cont == NULL ||
534         src == NULL || src->cont == NULL)
535     {
536         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
537                 __FILE__,__LINE__);
538         return -1;
539     }
540
541     if (dst->size < src->size)
542     {
543         if ((ptr=realloc(dst->cont,src->size*sizeof(double)))==NULL)
544         {
545             addError(CRITICAL,"Critical lack of memory in %s at line %d",
546                     __FILE__,__LINE__);
547             return -1;
548         }
549         else
550         {
551             dst->cont=ptr;
552             dst->size=src->size;
553         }
554     }
555
556     memcpy(dst->cont,src->cont,src->size*sizeof(double));
557     dst->top=src->top;
558
559     return 0;
560 }
```

4.7.4.2 int dblVecDestroy (DoubleVec *)

Definition at line 514 of file common.c.

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

```
515 {
516     if (vec == NULL || vec->cont == NULL)
517     {
518         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
519                 __FILE__,__LINE__);
520         return -1;
521     }
522
523     free(vec->cont);
524     free(vec);
525 }
```

```

526     return 0;
527 }

```

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()`.

```

498 {
499     if (vec == NULL || vec->cont == NULL)
500     {
501         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
502                 __FILE__, __LINE__);
503         return -1;
504     }
505
506     free(vec->cont);
507     vec->cont = NULL;
508     vec->size = 0;
509     vec->top = 0;
510
511     return 0;
512 }

```

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`.

```

641 {
642     if (vec == NULL || vec->cont == NULL || val == NULL)
643     {
644         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
645                 __FILE__, __LINE__);
646         return -1;
647     }
648
649     if (index < 0)
650     {
651         addError(CRITICAL, "Bad argument (index < 0) in %s at line %d",
652                 __FILE__, __LINE__);
653         return -1;
654     }
655
656     if (index >= vec->size)
657     {
658         addError(CRITICAL, "Bad argument (wrong index) in %s at line %d",
659                 __FILE__, __LINE__);
660         return -1;
661     }
662
663     *val = vec->cont[index];
664
665     return 0;
666
667 }

```

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().

```

470 {
471     void* ptr=NULL;
472
473     if (vec == NULL)
474     {
475         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
476                 __FILE__,__LINE__);
477         return -1;
478     }
479
480     if (size == -1)
481         size = DBLVEC_INITSIZE;
482
483     if ((ptr = calloc(size,sizeof(double))) == NULL)
484     {
485         addError(CRITICAL,"Critical lack of memory in %s at line %d",
486                 __FILE__,__LINE__);
487         return -1;
488     }
489
490     vec->size = size;
491     vec->top = 0;
492     vec->cont = ptr;
493
494     return 0;
495 }
```

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.

```

440 {
441     DoubleVec* vec=NULL;
442     void* ptr=NULL;
443
444     if ((vec = calloc(1,sizeof(DoubleVec))) == NULL)
445     {
446         addError(CRITICAL,"Critical lack of memory in %s at line %d",
447                 __FILE__,__LINE__);
448         return NULL;
449     }
450
451     if (size == -1)
452         size = DBLVEC_INITSIZE;
453
454     if ((ptr = calloc(size,sizeof(double))) == NULL)
455     {
456         addError(CRITICAL,"Critical lack of memory in %s at line %d",
457                 __FILE__,__LINE__);
458         free(vec);
459         return NULL;
460     }
```

```
460     }
461
462     vec->size = size;
463     vec->top = 0;
464     vec->cont = ptr;
465
466     return vec;
467 }
```

4.7.4.7 int dblVecPopBack (DoubleVec *, double *)

Definition at line 592 of file common.c.

References `addError()`, `DoubleVec::cont`, `CRITICAL`, and `DoubleVec::top`.

```
593 {
594     if (vec == NULL || vec->cont == NULL || val == NULL)
595     {
596         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
597                 __FILE__, __LINE__);
598         return -1;
599     }
600
601     if (vec->top == 0)
602     {
603         addError(CRITICAL, "Pop on empty stack in %s at line %d",
604                 __FILE__, __LINE__);
605         return -1;
606     }
607
608     *val = vec->cont[vec->top - 1];
609     vec->top--;
610
611     return 0;
612 }
```

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`.

```
563 {
564     void* ptr=NULL;
565
566     if (vec == NULL || vec->cont == NULL)
567     {
568         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
569                 __FILE__, __LINE__);
570         return -1;
571     }
572
573     if (vec->top >= vec->size)
574     {
575         if ((ptr = realloc(vec->cont, vec->size *
576                           2 * sizeof(double))) == NULL)
577         {
578             addError(CRITICAL, "Critical lack of memory in %s at line %d",
579                     __FILE__, __LINE__);
580             return -1;
581         }
582     }
```

```

582
583     vec->size *= 2;
584     vec->cont = ptr;
585 }
586
587 vec->cont[vec->top++] = val;
588
589 return 0;
590 }

```

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().

```

616 {
617     void* ptr=NULL;
618
619     if (vec == NULL || vec->cont == NULL)
620     {
621         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
622             __FILE__,__LINE__);
623         return -1;
624     }
625
626     if ((ptr = realloc(vec->cont, newsize*sizeof(double))) == NULL)
627     {
628         addError(CRITICAL,"Critical lack of memory in %s at line %d",
629             __FILE__,__LINE__);
630         return -1;
631     }
632
633     vec->cont = ptr;
634     memset(ptr+ (vec->size * sizeof(double)), 0, (newsize - vec->size)*sizeof(double));
635     vec->size = newsize;
636
637     return 0;
638 }

```

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.

```

670 {
671     if (vec == NULL || vec->cont == NULL)
672     {
673         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
674             __FILE__,__LINE__);
675         return -1;
676     }
677
678     if (index < 0)
679     {
680         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
681             __FILE__,__LINE__);

```

```

682     return -1;
683 }
684
685 if (index >= vec->size)
686 {
687     if (dblVecResize(vec,max(vec->size * 2,index+1))<0)
688     {
689         addError(CRITICAL,"Unable to resize double vector in %s at line %d",
690             __FILE__,__LINE__);
691         return -1;
692     }
693 }
694
695 vec->cont[index] = val;
696 vec->top = max(vec->top, index+1);
697
698 return 0;
699 }

```

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`.

```

282 {
283     if (list == NULL || list->cont == NULL)
284     {
285         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
286             __FILE__,__LINE__);
287         return -1;
288     }
289
290     if (index < 0)
291     {
292         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
293             __FILE__,__LINE__);
294         return -1;
295     }
296
297     if ((list->top >= list->size) || (index >= list->size))
298     {
299         if (longListResize(list,max(list->size * 2,index+1))<0)
300         {
301             addError(CRITICAL,"Unable to resize long vector in %s at line %d",
302                 __FILE__,__LINE__);
303             return -1;
304         }
305     }
306
307     if (index < list->top)
308     {
309         memmove(list->cont+index+1,list->cont+index,(list->top-index) * sizeof(long));
310     }
311     list->cont[index]=val;
312     list->top++;
313
314     return 0;
315 }

```

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().

```

358 {
359     int i=0,j=0;
360     LongList tmpList;
361
362     if (la == NULL || la->cont == NULL ||
363         lb == NULL || lb->cont == NULL ||
364         dest == NULL || dest->cont == NULL)
365     {
366         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
367             __FILE__,__LINE__);
368         return -1;
369     }
370
371     qsort(la->cont,la->top,sizeof(long),&longCompare);
372     qsort(lb->cont,lb->top,sizeof(long),&longCompare);
373
374     if (longListInit(&tmpList,la->top+lb->top)<0)
375     {
376         addError(CRITICAL,"Unable to initialize temporary list of longs in %s at line %d",
377             __FILE__,__LINE__);
378         return -1;
379     }
380
381     while ((i<la->top) || (j<lb->top))
382     {
383         if ((j==lb->top) || (i<la->top && la->cont[i]<lb->cont[j]))
384         {
385             longListPushBack(&tmpList,la->cont[i]);
386             i++;
387         }
388         else if ((i==la->top) || (j<lb->top && la->cont[i]>lb->cont[j]))
389         {
390             longListPushBack(&tmpList,lb->cont[j]);
391             j++;
392         }
393         else if (la->cont[i]==lb->cont[j])
394         {
395             longListPushBack(&tmpList,la->cont[i]);
396             i++;
397             j++;
398         }
399         else
400         {
401             addError(CRITICAL,"Internal error in %s at line %d",
402                 __FILE__,__LINE__);
403             longListEnd(&tmpList);
404             return -1;
405         }
406     }
407
408     if (longListCopy(dest,&tmpList)<0)
409     {
410         addError(CRITICAL,"Unable to create resulting merged list of longs in %s at line %d",
411             __FILE__,__LINE__);
412         longListEnd(&tmpList);
413         return -1;
414     }
415

```



```

416     longListEnd(&tmpList);
417
418     return 0;
419 }

```

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\(\)](#).

```

318 {
319     if (list == NULL || list->cont == NULL)
320     {
321         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
322             __FILE__,__LINE__);
323         return -1;
324     }
325
326     if (index < 0)
327     {
328         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
329             __FILE__,__LINE__);
330         return -1;
331     }
332
333     if (index >= list->top)
334     {
335         addError(WARNING,"Removing inexistent list element in %s at line %d",
336             __FILE__,__LINE__);
337         return -1;
338     }
339     else
340     {
341         memmove(list->cont+index,list->cont+index+1,(list->top-index-1) * sizeof(long));
342         list->top--;
343         return 0;
344     }
345 }

```

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\(\)](#).

```

421     {
422
423     if (list == NULL || list->cont == NULL)
424     {
425         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
426             __FILE__,__LINE__);
427         return -1;
428     }
429
430     qsort(list->cont,list->top,sizeof(long),&longCompare);
431

```

```
432     return 0;
433 }
```

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.

```
107 {
108     long *ptr=NULL;
109
110     if (dst == NULL || dst->cont == NULL ||
111         src == NULL || src->cont == NULL)
112     {
113         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
114                 __FILE__,__LINE__);
115         return -1;
116     }
117
118     if (dst->size < src->size)
119     {
120         if ((ptr=(long*) realloc(dst->cont,src->size*sizeof(long)))==NULL)
121         {
122             addError(CRITICAL,"Critical lack of memory in %s at line %d",
123                     __FILE__,__LINE__);
124             return -1;
125         }
126         else
127         {
128             dst->cont=ptr;
129             dst->size=src->size;
130         }
131     }
132
133     memcpy(dst->cont,src->cont,src->size*sizeof(long));
134     dst->top=src->top;
135
136     return 0;
137 }
```

4.7.4.16 int longVecDestroy (LongVec *)

Definition at line 91 of file common.c.

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

```
92 {
93     if (vec == NULL || vec->cont == NULL)
94     {
95         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
96                 __FILE__,__LINE__);
97         return -1;
98     }
99
100     free(vec->cont);
101     free(vec);
102
103     return 0;
104 }
```

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()`.

```

75 {
76     if (vec == NULL || vec->cont == NULL)
77     {
78         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
79                 __FILE__, __LINE__);
80         return -1;
81     }
82
83     free(vec->cont);
84     vec->cont = NULL;
85     vec->size = 0;
86     vec->top = 0;
87
88     return 0;
89 }
```

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()`.

```

217 {
218     if (vec == NULL || vec->cont == NULL || val == NULL)
219     {
220         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
221                 __FILE__, __LINE__);
222         return -1;
223     }
224
225     if (index < 0)
226     {
227         addError(CRITICAL, "Bad argument (index < 0) in %s at line %d",
228                 __FILE__, __LINE__);
229         return -1;
230     }
231
232     if (index >= vec->size)
233     {
234         addError(CRITICAL, "Bad argument (wrong index) in %s at line %d",
235                 __FILE__, __LINE__);
236         return -1;
237     }
238
239     *val = vec->cont[index];
240
241     return 0;
242
243 }
```

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().

```

47 {
48     void* ptr=NULL;
49
50     if (vec == NULL)
51     {
52         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
53                 __FILE__,__LINE__);
54         return -1;
55     }
56
57     if (size == -1)
58         size = LONGVEC_INITSIZE;
59
60     if ((ptr = calloc(size,sizeof(long))) == NULL)
61     {
62         addError(CRITICAL,"Critical lack of memory in %s at line %d",
63                 __FILE__,__LINE__);
64         return -1;
65     }
66
67     vec->size = size;
68     vec->top = 0;
69     vec->cont = ptr;
70
71     return 0;
72 }
```

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.

```

17 {
18     LongVec* vec=NULL;
19     void* ptr=NULL;
20
21     if ((vec = calloc(1,sizeof(LongVec))) == NULL)
22     {
23         addError(CRITICAL,"Critical lack of memory in %s at line %d",
24                 __FILE__,__LINE__);
25         return NULL;
26     }
27
28     if (size == -1)
29         size = LONGVEC_INITSIZE;
30
31     if ((ptr = calloc(size,sizeof(long))) == NULL)
32     {
33         addError(CRITICAL,"Critical lack of memory in %s at line %d",
34                 __FILE__,__LINE__);
35         free(vec);
36         return NULL;
37     }
```

```
37     }
38
39     vec->size = size;
40     vec->top = 0;
41     vec->cont = ptr;
42
43     return vec;
44 }
```

4.7.4.21 int longVecPopBack (LongVec *, long *)

Definition at line 169 of file common.c.

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

```
170 {
171     if (vec == NULL || vec->cont == NULL || val == NULL)
172     {
173         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
174                 __FILE__, __LINE__);
175         return -1;
176     }
177
178     if (vec->top == 0)
179     {
180         addError(CRITICAL, "Pop on empty stack in %s at line %d",
181                 __FILE__, __LINE__);
182         return -1;
183     }
184
185     *val = vec->cont[vec->top - 1];
186     vec->top--;
187
188     return 0;
189 }
```

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`.

```
140 {
141     void* ptr=NULL;
142
143     if (vec == NULL || vec->cont == NULL)
144     {
145         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
146                 __FILE__, __LINE__);
147         return -1;
148     }
149
150     if (vec->top >= vec->size)
151     {
152         if ((ptr = realloc(vec->cont, vec->size *
153                           2 * sizeof(long))) == NULL)
154         {
155             addError(CRITICAL, "Critical lack of memory in %s at line %d",
156                     __FILE__, __LINE__);
157             return -1;
158         }
159     }
```

```

159
160     vec->size *= 2;
161     vec->cont = ptr;
162 }
163
164 vec->cont[vec->top++] = val;
165
166 return 0;
167 }

```

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().

```

192 {
193     void* ptr=NULL;
194
195     if (vec == NULL || vec->cont == NULL)
196     {
197         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
198                 __FILE__, __LINE__);
199         return -1;
200     }
201
202     if ((ptr = realloc(vec->cont, newsize*sizeof(long))) == NULL)
203     {
204         addError(CRITICAL,"Critical lack of memory in %s at line %d",
205                 __FILE__, __LINE__);
206         return -1;
207     }
208
209     vec->cont = ptr;
210     memset(ptr+ (vec->size * sizeof(long)), 0, (newsize - vec->size)*sizeof(long));
211     vec->size = newsize;
212
213     return 0;
214 }

```

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().

```

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 }

```

4.7.4.25 void* mycalloc (size_t, size_t)

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 }

```

4.7.4.26 void myfree (void *)

Definition at line 741 of file common.c.

References allocatedMemory, and free.

```

742 {
743     if (*(long*)(ptr-sizeof(long))) == -1)
744         fprintf(stderr, "Warning already freed\n");
745
746     allocatedMemory--*(long*)(ptr-sizeof(long));
747     *(long*)(ptr-sizeof(long))=-1;
748     free(ptr-sizeof(long));
749 }

```

4.7.4.27 void* mymalloc (size_t)

Definition at line 715 of file common.c.

References allocatedMemory, and malloc.

Referenced by mycalloc().

```
716 {
717     void *ptr;
718
719     if ((ptr=malloc(sz+sizeof(long)))!=NULL)
720         return NULL;
721
722     allocatedMemory+=sz;
723     *((long*)ptr)=sz;
724
725     return (ptr+sizeof(long));
726 }
```

4.7.4.28 void* myrealloc (void *, size_t)

Definition at line 728 of file common.c.

References allocatedMemory, and realloc.

```
729 {
730     void *retptr;
731
732     if ((retptr=realloc(ptr+sizeof(long),sz+sizeof(long)))!=NULL)
733         return NULL;
734
735     allocatedMemory+=sz-(*((long*)(retptr)));
736     *((long*)retptr)=sz;
737
738     return (retptr+sizeof(long));
739 }
```

4.7.5 Variable Documentation

4.7.5.1 long [allocatedMemory](#)

Definition at line 86 of file common.h.

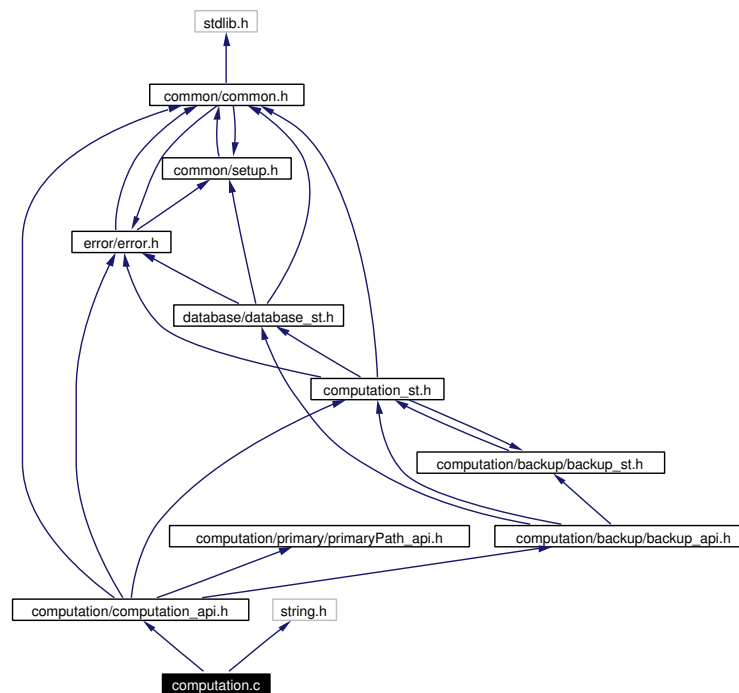
Referenced by myfree(), mymalloc(), and myrealloc().

4.8 computation.c File Reference

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

```
#include <string.h>
```

Include dependency graph for computation.c:



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.

```

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 }

```

4.8.1.2 int lspRequestDestroy ([LSPRequest](#) * req)

Definition at line 80 of file computation.c.

References addError(), CRITICAL, LSPRequest::forbidLinks, free, longListEnd, and LSPRequest::path.

```

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));
91     free(req);
92
93     return 0;
94 }

```

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().

```

97 {
98     if (req == NULL)
99     {
100         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
101                 __FILE__, __LINE__);
102         return -1;
103     }
104
105     longListEnd(&(req->forbidLinks));
106     longListEnd(&(req->path));
107
108     return 0;
109 }

```

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\(\)](#).

```

48 {
49     if (req == NULL)
50     {
51         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
52                 __FILE__, __LINE__);
53         return -1;
54     }
55
56     memset(req, 0, sizeof(LSPRequest));
57
58     if (longListInit(&(req->forbidLinks), -1) < 0)
59     {
60         addError(CRITICAL, "Unable to create label switched path request in %s at line %d",
61                 __FILE__, __LINE__);
62         return -1;
63     }
64
65     if (longListInit(&(req->path), -1) < 0)
66     {
67         longListEnd(&(req->forbidLinks));
68         addError(CRITICAL, "Unable to create label switched path request in %s at line %d",
69                 __FILE__, __LINE__);
70         return -1;
71     }
72
73     memset(req->bw, 0, NB_OA * sizeof(double));
74
75     req->rerouteInfo.id = -1;
76
77     return 0;
78 }

```

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](#).

```

185 {
186     long i;
187
188     if (reqList == NULL)
189     {
190         addError(CRITICAL, "LSPRequestList == NULL in %s at line %d",
191                 __FILE__, __LINE__);
192         return -1;
193     }
194
195     for (i=0; i<reqList->size; i++)
196     {
197         lspRequestEnd(&(reqList->cont[i]));
198     }
199
200     free(reqList->cont);
201
202     return 0;
203 }

```

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().

```

270 {
271     if (reqList == NULL)
272     {
273         addError(CRITICAL, "LSPRequestList == NULL in %s at line %d",
274                 __FILE__, __LINE__);
275         return NULL;
276     }
277
278     if (index < 0 || index >= reqList->size)
279     {
280         addError(CRITICAL, "Out of bound index in %s at line %d",
281                 __FILE__, __LINE__);
282         return NULL;
283     }
284
285     return &(reqList->cont[index]);
286 }

```

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, LSPREQUESTLIST_INITSIZE, lspRequestEnd(), lspRequestInit(), and LSPRequestList::size.

```

144 {
145     long i;
146
147     if (reqList == NULL)
148     {
149         addError(CRITICAL, "LSPRequestList == NULL in %s at line %d",
150                 __FILE__, __LINE__);
151         return -1;

```

```

152     }
153
154     size = (size<=0?LSPREQLIST_INITSIZE:size);
155
156     if ((reqList->cont = calloc(size, sizeof(LSPRequest))) == NULL)
157     {
158         addError(CRITICAL,"Impossible to allocate memory for LSPRequestList in %s at line %d",
159             __FILE__,__LINE__);
160         return -1;
161     }
162
163     for (i=0; i<size; i++)
164     {
165         if (lspRequestInit(&(reqList->cont[i])) < 0)
166         {
167             addError(CRITICAL,"Error while initialisation of LSPRequest in %s at line %d",
168                 __FILE__,__LINE__);
169             // clean up
170             for (i--;i>=0;i--)
171             {
172                 lspRequestEnd(&(reqList->cont[i]));
173             }
174             free(reqList->cont);
175             return -1;
176         }
177     }
178
179     reqList->size = size;
180
181     return 0;
182 }

```

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\(\)](#).

```

206 {
207     long i;
208
209     if (reqList == NULL)
210     {
211         addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
212             __FILE__,__LINE__);
213         return -1;
214     }
215
216     if (reqList->size < size)
217     {
218         LSPRequest* ptr;
219
220         if ((ptr = realloc(reqList->cont, size*sizeof(LSPRequest))) == NULL)
221         {
222             addError(CRITICAL,"Impossible to allocate memory for LSPRequestList in %s at line %d",
223                 __FILE__,__LINE__);
224             return -1;
225         }
226
227         reqList->cont = ptr;
228

```

```

229     for (i=reqList->size; i<size; i++)
230     {
231         if (lspRequestInit(&(reqList->cont[i])) < 0)
232         {
233             addError(CRITICAL,"Error while initialisation of LSPRequest in %s at line %d",
234                 __FILE__,__LINE__);
235             // clean up
236             for (i--;i>=reqList->size;i--)
237             {
238                 lspRequestEnd(&(reqList->cont[i]));
239             }
240             return -1;
241         }
242     }
243     reqList->size = size;
244 }
245 else
246 {
247     for (i=reqList->size-1; i>=size; i--)
248     {
249         lspRequestEnd(&(reqList->cont[i]));
250     }
251     reqList->size = size;
252 }
253
254 return 0;
255 }

```

4.8.1.9 int lspRequestListSet ([LSPRequestList](#) * *reqList*, [LSPRequest](#) * *req*, long *index*)

Definition at line 288 of file computation.c.

References [addError\(\)](#), and [CRITICAL](#).

```

289 {
290     addError(CRITICAL,"Trying to call an undefined function in %s at line %d",
291         __FILE__,__LINE__);
292
293     return 0;
294 }

```

4.8.1.10 long lspRequestListSize ([LSPRequestList](#) * *reqList*)

Definition at line 257 of file computation.c.

References [addError\(\)](#), [CRITICAL](#), and [LSPRequestList::size](#).

```

258 {
259     if (reqList == NULL)
260     {
261         addError(CRITICAL,"LSPRequestList == NULL in %s at line %d",
262             __FILE__,__LINE__);
263         return -1;
264     }
265
266     return reqList->size;
267 }

```

4.8.1.11 LSPRequest* lspRequestNew ()

Definition at line 12 of file computation.c.

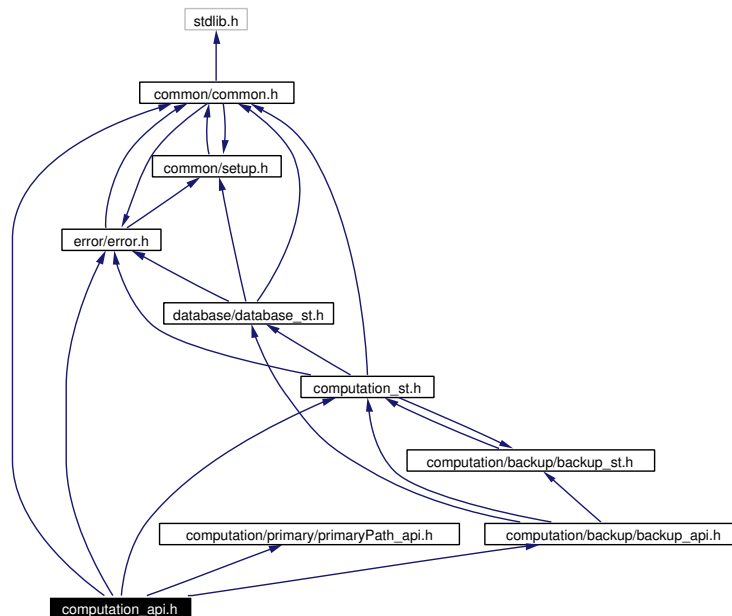
References `addError()`, `LSPRequest::bw`, `calloc`, `CRITICAL`, `LSPRequest::forbidLinks`, `free`, `LSPPrerouteInfo::id`, `longListEnd`, `longListInit`, `NB_OA`, `LSPRequest::path`, and `LSPRequest::rerouteInfo`.

```
13 {
14     LSPRequest* req;
15
16     if ((req=calloc(1,sizeof(LSPRequest)))==NULL)
17     {
18         addError(CRITICAL,"Critical lack of memory in %s at line %d",
19                 __FILE__,__LINE__);
20         return NULL;
21     }
22
23     if (longListInit(&(req->forbidLinks),-1)<0)
24     {
25         free(req);
26         addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
27                 __FILE__,__LINE__);
28         return NULL;
29     }
30
31     if (longListInit(&(req->path),-1)<0)
32     {
33         longListEnd(&(req->forbidLinks));
34         free(req);
35         addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
36                 __FILE__,__LINE__);
37         return NULL;
38     }
39
40     memset(req->bw, 0, NB_OA * sizeof(double));
41
42     req->rerouteInfo.id = -1;
43
44     return req;
45 }
```

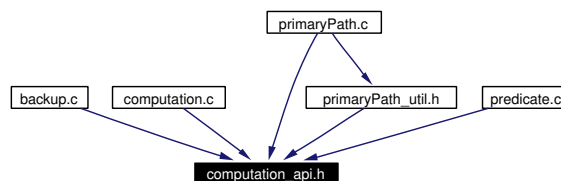
4.9 computation_api.h File Reference

```
#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));

```

```

91     free(req);
92
93     return 0;
94 }

```

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 lspRequestListResize().

```

97 {
98     if (req == NULL)
99     {
100         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
101                 __FILE__, __LINE__);
102         return -1;
103     }
104
105     longListEnd(&(req->forbidLinks));
106     longListEnd(&(req->path));
107
108     return 0;
109 }

```

4.9.1.4 int lspRequestInit (LSPRequest *)

Definition at line 47 of file computation.c.

References addError(), LSPRequest::bw, CRITICAL, LSPRequest::forbidLinks, LSPPrerouteInfo::id, longListEnd, longListInit, NB_OA, LSPRequest::path, and LSPRequest::rerouteInfo.

Referenced by computeCost(), isValidLSPLink(), lspRequestListInit(), and lspRequestListResize().

```

48 {
49     if (req == NULL)
50     {
51         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
52                 __FILE__, __LINE__);
53         return -1;
54     }
55
56     memset(req, 0, sizeof(LSPRequest));
57
58     if (longListInit(&(req->forbidLinks), -1) < 0)
59     {
60         addError(CRITICAL, "Unable to create label switched path request in %s at line %d",
61                 __FILE__, __LINE__);
62         return -1;
63     }
64
65     if (longListInit(&(req->path), -1) < 0)
66     {
67         longListEnd(&(req->forbidLinks));
68         addError(CRITICAL, "Unable to create label switched path request in %s at line %d",
69                 __FILE__, __LINE__);
70         return -1;
71     }

```

```

72
73     memset(req->bw, 0, NB_OA * sizeof(double));
74
75     req->rerouteInfo.id = -1;
76
77     return 0;
78 }

```

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`.

```

185 {
186     long i;
187
188     if (reqList == NULL)
189     {
190         addError(CRITICAL, "LSPRequestList == NULL in %s at line %d",
191                 __FILE__, __LINE__);
192         return -1;
193     }
194
195     for (i=0; i<reqList->size; i++)
196     {
197         lspRequestEnd(&(reqList->cont[i]));
198     }
199
200     free(reqList->cont);
201
202     return 0;
203 }

```

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()`.

```

270 {
271     if (reqList == NULL)
272     {
273         addError(CRITICAL, "LSPRequestList == NULL in %s at line %d",
274                 __FILE__, __LINE__);
275         return NULL;
276     }
277
278     if (index < 0 || index >= reqList->size)
279     {
280         addError(CRITICAL, "Out of bound index in %s at line %d",
281                 __FILE__, __LINE__);
282         return NULL;
283     }
284
285     return &(reqList->cont[index]);
286 }

```

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.

```

144 {
145     long i;
146
147     if (reqList == NULL)
148     {
149         addError(CRITICAL, "LSPRequestList == NULL in %s at line %d",
150             __FILE__, __LINE__);
151         return -1;
152     }
153
154     size = (size <= 0 ? LSPREQLIST_INITSIZE : size);
155
156     if ((reqList->cont = calloc(size, sizeof(LSPRequest))) == NULL)
157     {
158         addError(CRITICAL, "Impossible to allocate memory for LSPRequestList in %s at line %d",
159             __FILE__, __LINE__);
160         return -1;
161     }
162
163     for (i=0; i<size; i++)
164     {
165         if (lspRequestInit(&(reqList->cont[i])) < 0)
166         {
167             addError(CRITICAL, "Error while initialisation of LSPRequest in %s at line %d",
168                 __FILE__, __LINE__);
169             // clean up
170             for (i--; i>=0; i--)
171             {
172                 lspRequestEnd(&(reqList->cont[i]));
173             }
174             free(reqList->cont);
175             return -1;
176         }
177     }
178
179     reqList->size = size;
180
181     return 0;
182 }
```

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().

```

206 {
207     long i;
208
209     if (reqList == NULL)
210     {
211         addError(CRITICAL, "LSPRequestList == NULL in %s at line %d",
212             __FILE__, __LINE__);

```

```

213     return -1;
214 }
215
216 if (reqList->size < size)
217 {
218     LSPRequest* ptr;
219
220     if ((ptr = realloc(reqList->cont, size*sizeof(LSPRequest))) == NULL)
221     {
222         addError(CRITICAL, "Impossible to allocate memory for LSPRequestList in %s at line %d",
223                 __FILE__, __LINE__);
224         return -1;
225     }
226
227     reqList->cont = ptr;
228
229     for (i=reqList->size; i<size; i++)
230     {
231         if (lspRequestInit(&(reqList->cont[i])) < 0)
232         {
233             addError(CRITICAL, "Error while initialisation of LSPRequest in %s at line %d",
234                     __FILE__, __LINE__);
235             // clean up
236             for (i--; i>=reqList->size; i--)
237             {
238                 lspRequestEnd(&(reqList->cont[i]));
239             }
240             return -1;
241         }
242     }
243     reqList->size = size;
244 }
245 else
246 {
247     for (i=reqList->size-1; i>=size; i--)
248     {
249         lspRequestEnd(&(reqList->cont[i]));
250     }
251     reqList->size = size;
252 }
253
254 return 0;
255 }

```

4.9.1.9 int lspRequestListSet (LSPRequestList *, LSPRequest *, long)

Definition at line 288 of file computation.c.

References `addError()`, and `CRITICAL`.

```

289 {
290     addError(CRITICAL, "Trying to call an undefined function in %s at line %d",
291             __FILE__, __LINE__);
292
293     return 0;
294 }

```

4.9.1.10 long lspRequestListsize (LSPRequestList *)

4.9.1.11 LSPRequest* lspRequestNew ()

Definition at line 12 of file computation.c.

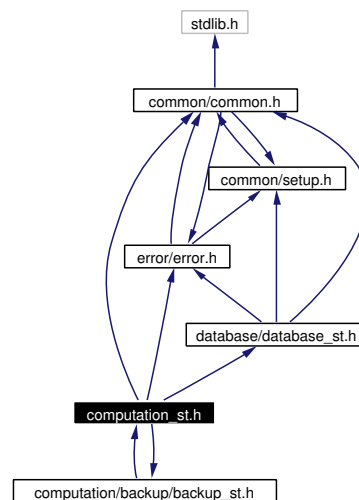
References addError(), LSPRequest::bw, calloc, CRITICAL, LSPRequest::forbidLinks, free, LSPPrerouteInfo::id, longListEnd, longListInit, NB_OA, LSPRequest::path, and LSPRequest::rerouteInfo.

```
13 {
14     LSPRequest* req;
15
16     if ((req=calloc(1,sizeof(LSPRequest)))==NULL)
17     {
18         addError(CRITICAL,"Critical lack of memory in %s at line %d",
19                 __FILE__,__LINE__);
20         return NULL;
21     }
22
23     if (longListInit(&(req->forbidLinks),-1)<0)
24     {
25         free(req);
26         addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
27                 __FILE__,__LINE__);
28         return NULL;
29     }
30
31     if (longListInit(&(req->path),-1)<0)
32     {
33         longListEnd(&(req->forbidLinks));
34         free(req);
35         addError(CRITICAL,"Unable to create label switched path request in %s at line %d",
36                 __FILE__,__LINE__);
37         return NULL;
38     }
39
40     memset(req->bw, 0, NB_OA * sizeof(double));
41
42     req->rerouteInfo.id = -1;
43
44     return req;
45 }
```

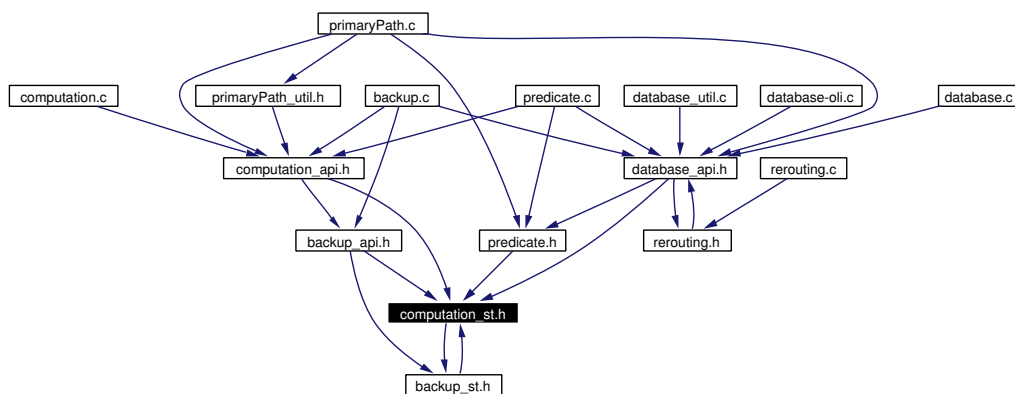
4.10 computation_st.h File Reference

```
#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](#)

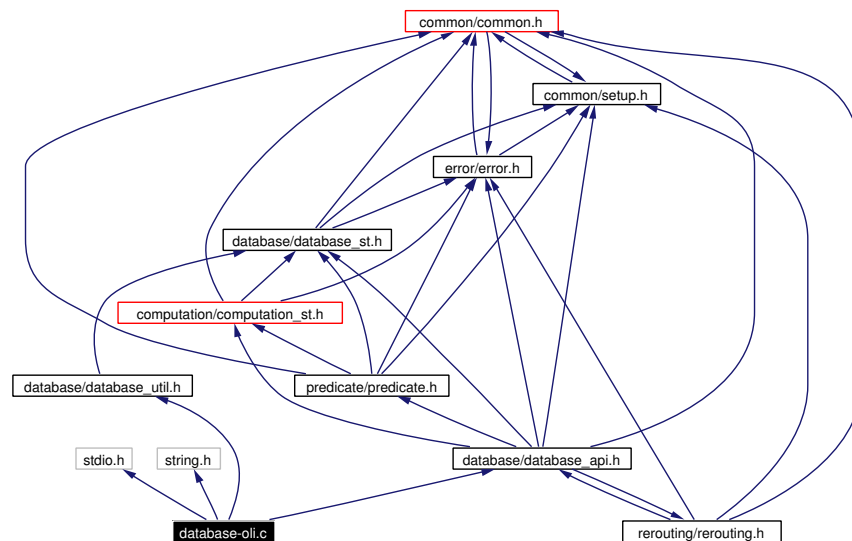
Rerouting Information structure.

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 resources of the preempted one can be used. But it is also interesting to take into account the link where the preemption occurred, 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;
```

```

743         else
744             seenLinks++;
745
746         totBbw = 0;
747         totFbw = 0;
748
749         for(p=0; p<NB_PREEMPTION; ++p)
750         {
751             if (i < bbw[oa][p].size)
752                 totBbw += bbw[oa][p].cont[i];
753             if (i < fbw[oa][p].size)
754                 totFbw += fbw[oa][p].cont[i];
755
756             gbw[i].cont[p] = max(0, totBbw - totFbw);
757         }
758     }
759
760     seenNodes = 0;
761
762     // phase 1b (nodes or any set of links)
763     for (i=0; seenNodes<dataBase->nbNodes; ++i)
764     {
765         if (dataBase->nodeVec.cont[i] == NULL)
766             continue;
767         else
768             seenNodes++;
769
770         totBbw = 0;
771         totFbw = 0;
772
773         for(p=0; p<NB_PREEMPTION; ++p)
774         {
775             LongList* lst;
776             if ((lst = DBgetNodeInNeighb(dataBase, i)) != NULL)
777             {
778                 int l;
779                 for (l=0; l<lst->top; ++l)
780                 {
781                     int lnkID = DBgetLinkID(dataBase, lst->cont[l], i);
782                     if (lnkID < bbw[oa][p].size)
783                         totBbw += bbw[oa][p].cont[lnkID];
784                     if (lnkID < fbw[oa][p].size)
785                         totFbw += fbw[oa][p].cont[lnkID];
786                 }
787             }
788
789             gbw[i + nbLink].cont[p] = max(0, totBbw - totFbw);
790         }
791     }
792
793     // phase 2
794     oldM = 0;
795
796     for (p=0; p<NB_PREEMPTION; ++p)
797     {
798         m = 0;
799         for (i=0; i<nbLink + nbNode; ++i)
800         {
801             if (gbw[i].cont[p] > m)
802             {
803                 m = gbw[i].cont[p];
804             }
805         }
806
807         rbw[oa][p] = pbw[oa][p] + m - oldM;
808         oldM = m;
809     }

```

```

810     }
811
812     for (i=0; i<nbLink + nbNode; ++i)
813         dblVecEnd(&(gbw[i]));
814
815     free(gbw);
816 */
817
818     for (oa=0; oa<NB_OA; ++oa)
819         for (p=0; p<NB_PREEMPTION; ++p)
820             {
821                 rbw[oa][p] = pbw[oa][p];
822             }
823
824 #if defined LINUX && defined TIME1
825     gettimeofday(&t2, &tz);
826     fprintf(stderr, "Time to compute rbw : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
827         (t2.tv_usec - t1.tv_usec) / 1000.0);
828 #endif
829
830     return 0;
831
832 }

```

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.

```

1556 {
1557     DBLink* link=NULL;
1558     int ret=0;
1559
1560     if (dataBase == NULL || initLinkState==NULL
1561         || id < 0 || src < 0 || dst < 0)
1562     {
1563         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1564             __FILE__, __LINE__);
1565         return -1;
1566     }
1567
1568     if (((id<dataBase->linkSrcVec.size) && (dataBase->linkSrcVec.cont[id]>0))
1569         ||
1570         ((id<dataBase->linkDstVec.size) && (dataBase->linkDstVec.cont[id]>0)))
1571     {
1572         addError(CRITICAL,"Trying to add a link with a reserved ID (ID=%ld) in %s at line %d",
1573             id, __FILE__, __LINE__);
1574         return -1;
1575     }
1576
1577     if ((link = DBlinkNew()) == NULL)
1578     {
1579         addError(CRITICAL,"Unable to create link in %s at line %d",
1580             __FILE__, __LINE__);
1581         return -1;
1582     }
1583
1584     link->id=id;
1585
1586     if (DBlinkStateCopy(&(link->state), initLinkState))
1587     {
1588         addError(CRITICAL,"Unable to create link in %s at line %d",
1589             __FILE__, __LINE__);
1590         DBlinkDestroy(link);

```

```

1591         return -1;
1592     }
1593
1594     if ((DBnodeVecGet(&(dataBase->nodeVec),src) == NULL) ||
1595         (DBnodeVecGet(&(dataBase->nodeVec),dst) == NULL))
1596     {
1597         addError(CRITICAL,"Source or destination node doesn't exist in %s at line %d",
1598             __FILE__,__LINE__);
1599         DBlinkDestroy(link);
1600         return -1;
1601     }
1602
1603     if (DBlinkTabSet(&(dataBase->linkTab),link,src,dst)<0)
1604     {
1605         addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %d",
1606             __FILE__,__LINE__);
1607         DBlinkDestroy(link);
1608         return -1;
1609     }
1610
1611     ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
1612     ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[dst]->inNeighb),src));
1613
1614     ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[src]->outNeighb)));
1615     ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[dst]->inNeighb)));
1616
1617     ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,src+1));
1618     ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,dst+1));
1619
1620     // Maximum non-null element
1621     dataBase->linkSrcVec.top = max(dataBase->linkSrcVec.top, id+1);
1622     dataBase->linkDstVec.top = dataBase->linkSrcVec.top;
1623
1624     if (ret<0)
1625     {
1626         addError(CRITICAL,"Link addition uncomplete in %s at line %d",
1627             __FILE__,__LINE__);
1628     }
1629
1630     dataBase->nbLinks++;
1631
1632     return ret;
1633 }

```

4.11.3.3 int DBaddLSP (DataBase * dataBase, DBLabelSwitchedPath * lsp, LongList * preemptList)

Definition at line 1679 of file database-oli.c.

```

1680 {
1681     DBLabelSwitchedPath *newLSP, *contentLSP=NULL;
1682     DBLSPList *lspList;
1683     int i,ret=0;
1684     DBLink *lnk=NULL;
1685     LongVec isProcessed;
1686     double rerouteGain[NB_OA];
1687     bool allowLSP=TRUE;
1688     #if defined SIMULATOR
1689         LongList idList;
1690     #elif defined AGENT
1691         int j;
1692         bool inPath=FALSE;
1693     #endif
1694

```

```

1695 #if defined LINUX && defined TIME2
1696     struct timezone tz;
1697     struct timeval t1,t2;
1698 #endif
1699
1700     if (dataBase == NULL || lsp==NULL)
1701     {
1702         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1703             __FILE__,__LINE__);
1704         return -1;
1705     }
1706
1707 #if defined LINUX && defined TIME2
1708     gettimeofday(&t1, &tz);
1709 #endif
1710
1711     if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
1712     {
1713         addError(CRITICAL,"Unable to initialize vector of longs in %s at line %d",
1714             __FILE__,__LINE__);
1715         return -1;
1716     }
1717
1718     memset(rerouteGain,0,NB_OA*sizeof(double));
1719
1720     // Check if establishment is possible
1721 #if defined SIMULATOR
1722     if (longListInit(&(idList),-1)<0)
1723     {
1724         addError(CRITICAL,"Unable to initialize list of longs in %s at line %d",
1725             __FILE__,__LINE__);
1726         return -1;
1727     }
1728     for (i=0;(i<lsp->path.top-1) && allowLSP;i++)
1729     {
1730         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
1731             lsp->path.cont[i+1]);
1732         allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
1733             &(lnk->state),lsp,rerouteGain);
1734         if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
1735         {
1736             addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1737                 __FILE__,__LINE__);
1738             longListEnd(&(idList));
1739             longVecEnd(&(isProcessed));
1740             return -1;
1741         }
1742         idList.top=0;
1743         if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
1744         {
1745             addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1746                 __FILE__,__LINE__);
1747             longListEnd(&(idList));
1748             longVecEnd(&(isProcessed));
1749             return -1;
1750         }
1751         if (longListMerge(&(idList),preemptList,preemptList)<0)
1752         {
1753             addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
1754                 __FILE__,__LINE__);
1755             longListEnd(&(idList));
1756             longVecEnd(&(isProcessed));
1757             return -1;
1758         }
1759         isProcessed.cont[lnk->id] = 1;
1760     }
1761     if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))

```

```

1762     {
1763         for (i=0;(i<lsp->primPath.top-1) && allowLSP;i++)
1764         {
1765             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
1766                             lsp->primPath.cont[i+1]);
1767             if (isProcessed.cont[lnk->id] == 0)
1768             {
1769                 allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1],
1770                                                         &(lnk->state),lsp,rerouteGain);
1771                 if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==
1772                     {
1773                     addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1774                               __FILE__,__LINE__);
1775                     longListEnd(&(idList));
1776                     longVecEnd(&(isProcessed));
1777                     return -1;
1778                 }
1779                 idList.top=0;
1780                 if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
1781                 {
1782                     addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1783                               __FILE__,__LINE__);
1784                     longListEnd(&(idList));
1785                     longVecEnd(&(isProcessed));
1786                     return -1;
1787                 }
1788                 if (longListMerge(&(idList),preemptList,preemptList)<0)
1789                 {
1790                     addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
1791                               __FILE__,__LINE__);
1792                     longListEnd(&(idList));
1793                     longVecEnd(&(isProcessed));
1794                     return -1;
1795                 }
1796                 isProcessed.cont[lnk->id] = 1;
1797             }
1798         }
1799     }
1800     longListEnd(&(idList));
1801 #elif defined AGENT
1802     for (i=0;(i<lsp->path.top-1) && (lsp->path.cont[i]!=dataBase->id);i++);
1803
1804     if (i<lsp->path.top-1)
1805     {
1806         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
1807                         lsp->path.cont[i+1]);
1808         allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
1809                                             &(lnk->state),lsp,rerouteGain);
1810         if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
1811         {
1812             addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1813                       __FILE__,__LINE__);
1814             longVecEnd(&(isProcessed));
1815         }
1816         if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)<0)
1817         {
1818             addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1819                       __FILE__,__LINE__);
1820             longVecEnd(&(isProcessed));
1821             return -1;
1822         }
1823         isProcessed.cont[lnk->id] = 1;
1824         inPath=TRUE;
1825     }
1826     if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
1827     {
1828         for (j=0;(j<lsp->primPath.top-1) && (lsp->primPath.cont[j]!=dataBase->id);j++);

```



```

1829
1830     if (j<lsp->primPath.top-1)
1831     {
1832         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[j],
1833                         lsp->primPath.cont[j+1]);
1834         if (isProcessed.cont[lnk->id] == 0)
1835         {
1836             allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[j],lsp->primPath.cont[j+1],
1837                                                    &(lnk->state),lsp,rerouteGain);
1838             if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==0)
1839             {
1840                 addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1841                           __FILE__,__LINE__);
1842                 longVecEnd(&(isProcessed));
1843             }
1844             if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList))
1845             {
1846                 addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1847                           __FILE__,__LINE__);
1848                 longVecEnd(&(isProcessed));
1849                 return -1;
1850             }
1851             isProcessed.cont[lnk->id] = 1;
1852         }
1853         inPath=TRUE;
1854     }
1855 }
1856 if (!inPath)
1857 {
1858     addError(CRITICAL,"Agent not concerned by this LSP in %s at line %d",
1859             __FILE__,__LINE__);
1860     longVecEnd(&(isProcessed));
1861     return -1;
1862 }
1863 #else
1864 // Generate an error;
1865 COMPILE_ERROR;
1866 #endif
1867
1868 if (!allowLSP)
1869 {
1870     addError(CRITICAL,"LSP refused by the predicate in %s at line %d",
1871             __FILE__,__LINE__);
1872     longVecEnd(&(isProcessed));
1873     return -1;
1874 }
1875
1876
1877 if ((newLSP=DBlspNew())==NULL)
1878 {
1879     addError(CRITICAL,"Unable to create LSP in %s at line %d",
1880             __FILE__,__LINE__);
1881     longVecEnd(&(isProcessed));
1882     return -1;
1883 }
1884
1885 if (DBlspCopy(newLSP,lsp)<0)
1886 {
1887     addError(CRITICAL,"Unable to create a valid LSP copy in %s at line %d",
1888             __FILE__,__LINE__);
1889     DBlspDestroy(newLSP);
1890     longVecEnd(&(isProcessed));
1891     return -1;
1892 }
1893
1894 if (DBlspVecSet(&(dataBase->lspVec),newLSP,newLSP->id)<0)
1895 {

```

```

1896         addError(CRITICAL,"Unable to insert LSP in the general LSP container in %s at line %d",
1897             __FILE__,__LINE__);
1898         DBlspDestroy(newLSP);
1899         longVecEnd(&(isProcessed));
1900         return -1;
1901     }
1902
1903     if (newLSP->noContentionId>=0)
1904     {
1905         if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),newLSP->noContentionId))==NULL)
1906         {
1907             addError(WARNING,"Unable to get no contention LSP in %s at line %d",
1908                 __FILE__,__LINE__);
1909             newLSP->noContentionId=-1;
1910             // not critical enough to abort
1911         }
1912         else
1913         {
1914             contentLSP->noContentionId=newLSP->id;
1915         }
1916     }
1917
1918     for (i=0;i<isProcessed.size;i++)
1919     {
1920         isProcessed.cont[i]=0;
1921     }
1922
1923
1924 #if defined SIMULATOR
1925     // Add the LSP to each link list and update all the linkstates (only once !!!!!)
1926     for (i=0;i<newLSP->path.top-1;i++)
1927     {
1928         lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
1929             newLSP->path.cont[i+1]);
1930         ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
1931         ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->path.cont[i],
1932             newLSP->path.cont[i+1], &(lnk->state), newLSP));
1933         isProcessed.cont[lnk->id] = 1;
1934     }
1935     if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
1936     {
1937         for (i=0;i<newLSP->primPath.top-1;i++)
1938         {
1939             lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
1940                 newLSP->primPath.cont[i+1]);
1941             if (isProcessed.cont[lnk->id] == 0)
1942             {
1943                 ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->primPath.cont[i],
1944                     newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
1945                 isProcessed.cont[lnk->id] = 1;
1946             }
1947         }
1948     }
1949 #elif defined AGENT
1950     // Add the LSP to the link attached to the agent and update the linkstate
1951     for (i=0;i<newLSP->path.top-1;i++)
1952     {
1953         lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
1954             newLSP->path.cont[i+1]);
1955         ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
1956
1957         if (newLSP->path.cont[i] == dataBase->id)
1958         {
1959             ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->path.cont[i],
1960                 newLSP->path.cont[i+1], &(lnk->state), newLSP));
1961             isProcessed.cont[lnk->id] = 1;
1962         }

```

```

1963     }
1964     if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
1965     {
1966         for (i=0;i<newLSP->primPath.top-1;i++)
1967         {
1968             lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
1969                             newLSP->primPath.cont[i+1]);
1970
1971             if (newLSP->primPath.cont[i] == dataBase->id)
1972             {
1973                 if (isProcessed.cont[lnk->id] == 0)
1974                 {
1975                     ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->primPath.cont[i],
1976                                                     newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
1977                 }
1978                 break;
1979             }
1980         }
1981     }
1982 #else
1983     // Generate an error;
1984     COMPILER_ERROR;
1985 #endif
1986
1987     longVecEnd(&(isProcessed));
1988
1989 #if defined LINUX && defined TIME2
1990     gettimeofday(&t2, &tz);
1991     fprintf(stderr, "Time to add a new LSP : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
1992             (t2.tv_usec - t1.tv_usec) / 1000.0);
1993 #endif
1994
1995
1996     if (ret<0)
1997     {
1998         addError(CRITICAL,"LSP addition uncomplete in %s at line %d",
1999                 __FILE__, __LINE__);
2000     }
2001
2002     return ret;
2003 }

```

4.11.3.4 int DBaddNode (DataBase * dataBase, long id)

Definition at line 1466 of file database-oli.c.

```

1467 {
1468     DBNode *node=NULL;
1469
1470     if (dataBase == NULL)
1471     {
1472         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1473                 __FILE__, __LINE__);
1474         return -1;
1475     }
1476
1477     if ((node=DBnodeNew()) == NULL)
1478     {
1479         addError(CRITICAL,"Unable to create node in %s at line %d",
1480                 __FILE__, __LINE__);
1481         return -1;
1482     }
1483
1484     node->id=id;

```

```

1485
1486     if (DBnodeVecSet(&(dataBase->nodeVec),node,id) < 0)
1487     {
1488         addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %d",
1489             __FILE__,__LINE__);
1490         DBnodeDestroy(node);
1491         return -1;
1492     }
1493
1494     dataBase->nbNodes++;
1495
1496     return 0;
1497 }

```

4.11.3.5 int DBdestroy (**DataBase** * *dataBase*)

Definition at line 1349 of file database-oli.c.

```

1350 {
1351     if (dataBase == NULL)
1352     {
1353         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1354             __FILE__,__LINE__);
1355         return -1;
1356     }
1357
1358     DBnodeVecEnd(&(dataBase->nodeVec));
1359     DBlspVecEnd(&(dataBase->lspVec));
1360     DBlinkTabEnd(&(dataBase->linkTab));
1361     longVecEnd(&(dataBase->linkSrcVec));
1362     longVecEnd(&(dataBase->linkDstVec));
1363
1364     free(dataBase);
1365
1366     return 0;
1367 }

```

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.

```

1260 {
1261     return evalLS(dataBase, src, dst, newLS, oldLS, req, REMOVE);
1262 }

```

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.

```

1254 {
1255     return evalLS(dataBase, src, dst, newLS, oldLS, req, SETUP);
1256 }

```

4.11.3.8 long DBgetID ([DataBase](#) * *dataBase*)

Definition at line 1369 of file database-oli.c.

```
1370 {
1371     if (dataBase == NULL)
1372     {
1373         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1374             __FILE__,__LINE__);
1375         return -1;
1376     }
1377     return dataBase->id;
1378 }
1379 }
```

4.11.3.9 long DBgetLinkDst ([DataBase](#) * *dataBase*, long *id*)

Definition at line 1421 of file database-oli.c.

```
1422 {
1423     long ret;
1424
1425     if (dataBase == NULL)
1426     {
1427         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1428             __FILE__,__LINE__);
1429         return -1;
1430     }
1431
1432     if (longVecGet(&(dataBase->linkDstVec),id,&ret)<0)
1433     {
1434         addError(CRITICAL,"Inexistent link in %s at line %d",
1435             __FILE__,__LINE__);
1436         return -1;
1437     }
1438
1439     return (ret-1);
1440 }
```

4.11.3.10 long DBgetLinkID ([DataBase](#) * *dataBase*, long *src*, long *dst*)

Definition at line 1381 of file database-oli.c.

```
1382 {
1383     DBLink *lnk=NULL;
1384
1385     if (dataBase == NULL || src < 0 || dst < 0)
1386     {
1387         addError(CRITICAL,"Bad argument (NULL or negative value) in %s at line %d",
1388             __FILE__,__LINE__);
1389         return -1;
1390     }
1391
1392     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst))==NULL)
1393     {
1394         return -1;
1395     }
1396
1397     return lnk->id;
1398 }
```

4.11.3.11 DBLSPList* DBgetLinkLSPs (DataBase * *dataBase*, long *src*, long *dst*)

Definition at line 2138 of file database-oli.c.

```

2139 {
2140     DBLink *lnk=NULL;
2141
2142     if (dataBase == NULL)
2143     {
2144         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2145             __FILE__,__LINE__);
2146         return NULL;
2147     }
2148
2149     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2150     {
2151         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2152             src,dst,__FILE__,__LINE__);
2153         return NULL;
2154     }
2155
2156     return &(lnk->lspList);
2157 }

```

4.11.3.12 long DBgetLinkSrc (DataBase * *dataBase*, long *id*)

Definition at line 1400 of file database-oli.c.

```

1401 {
1402     long ret;
1403
1404     if (dataBase == NULL)
1405     {
1406         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1407             __FILE__,__LINE__);
1408         return -1;
1409     }
1410
1411     if (longVecGet(&(dataBase->linkSrcVec),id,&ret)<0)
1412     {
1413         addError(CRITICAL,"Inexistent link in %s at line %d",
1414             __FILE__,__LINE__);
1415         return -1;
1416     }
1417
1418     return (ret-1);
1419 }

```

4.11.3.13 DBLinkState* DBgetLinkState (DataBase * *dataBase*, long *src*, long *dst*)

Definition at line 2159 of file database-oli.c.

```

2160 {
2161     DBLink *lnk=NULL;
2162
2163     if (dataBase == NULL)
2164     {
2165         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2166             __FILE__,__LINE__);

```

```

2167         return NULL;
2168     }
2169
2170     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2171     {
2172         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2173             src,dst,__FILE__,__LINE__);
2174         return NULL;
2175     }
2176
2177     return &(lnk->state);
2178 }

```

4.11.3.14 DBLabelSwitchedPath* DBgetLSP (DataBase * *dataBase*, long *id*)

Definition at line 2125 of file database-oli.c.

```

2126 {
2127     if (dataBase == NULL)
2128     {
2129         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2130             __FILE__,__LINE__);
2131         return NULL;
2132     }
2133
2134     return DBlspVecGet(&(dataBase->lspVec), id);
2135 }

```

4.11.3.15 long DBgetMaxNodeID (DataBase * *dataBase*)

Definition at line 1454 of file database-oli.c.

```

1455 {
1456     if (dataBase == NULL)
1457     {
1458         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1459             __FILE__,__LINE__);
1460         return -1;
1461     }
1462
1463     return dataBase->nodeVec.top-1;
1464 }

```

4.11.3.16 long DBgetNbLinks (DataBase * *dataBase*)

Definition at line 1543 of file database-oli.c.

```

1544 {
1545     if (dataBase == NULL)
1546     {
1547         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1548             __FILE__,__LINE__);
1549         return -1;
1550     }
1551
1552     return dataBase->nbLinks;
1553 }

```

4.11.3.17 long DBgetNbNodes (**DataBase** * *dataBase*)

Definition at line 1442 of file database-oli.c.

```
1443 {
1444     if (dataBase == NULL)
1445     {
1446         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1447             __FILE__,__LINE__);
1448         return -1;
1449     }
1450
1451     return dataBase->nbNodes;
1452 }
```

4.11.3.18 **LongList*** DBgetNodeInNeighb (**DataBase** * *dataBase*, long *id*)

Definition at line 2209 of file database-oli.c.

```
2210 {
2211     DBNode *node=NULL;
2212
2213     if (dataBase == NULL)
2214     {
2215         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2216             __FILE__,__LINE__);
2217         return NULL;
2218     }
2219
2220     if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
2221     {
2222         addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
2223             id,__FILE__,__LINE__);
2224         return NULL;
2225     }
2226
2227     return (&(node->inNeighb));
2228 }
```

4.11.3.19 **LongList*** DBgetNodeOutNeighb (**DataBase** * *dataBase*, long *id*)

Definition at line 2231 of file database-oli.c.

```
2232 {
2233     DBNode *node=NULL;
2234
2235     if (dataBase == NULL)
2236     {
2237         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2238             __FILE__,__LINE__);
2239         return NULL;
2240     }
2241
2242     if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
2243     {
2244         addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
2245             id,__FILE__,__LINE__);
2246         return NULL;
2247     }
2248 }
```



```

2248
2249     return (&(node->outNeighb));
2250 }

```

4.11.3.20 int DBlinkStateCopy (DBLinkState *dst, DBLinkState *src)

Definition at line 660 of file database-oli.c.

```

661 {
662     int i,j,ret=0;
663
664     if (dst == NULL || src == NULL)
665     {
666         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
667                 __FILE__, __LINE__);
668         return -1;
669     }
670
671     dst->color=src->color;
672     memcpy(&(dst->cap),&(src->cap),NB_OA * sizeof(double));
673     memcpy(&(dst->rbw),&(src->rbw),NB_OA * NB_PREEMPTION * sizeof(double));
674     memcpy(&(dst->pbw),&(src->pbw),NB_OA * NB_PREEMPTION * sizeof(double));
675
676     for (i=0;(i<NB_OA && ret>=0);i++)
677         for (j=0;(j<NB_PREEMPTION && ret>=0);j++)
678         {
679             ANDERROR(ret,dblVecCopy(&(dst->bbw[i][j]),&(src->bbw[i][j])));
680             ANDERROR(ret,dblVecCopy(&(dst->remoteBbw[i][j]),&(src->remoteBbw[i][j])));
681             ANDERROR(ret,dblVecCopy(&(dst->fbw[i][j]),&(src->fbw[i][j])));
682             ANDERROR(ret,dblVecCopy(&(dst->remoteFbw[i][j]),&(src->remoteFbw[i][j])));
683         }
684
685     if (ret<0)
686     {
687         addError(CRITICAL,"Link state copy uncomplete in %s at line %d",
688                 __FILE__, __LINE__);
689     }
690
691     return ret;
692 }

```

4.11.3.21 int DBlinkStateDestroy (DBLinkState *ls)

Definition at line 613 of file database-oli.c.

```

614 {
615     int i,j;
616
617     if (ls == NULL)
618     {
619         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
620                 __FILE__, __LINE__);
621         return -1;
622     }
623
624     for (i=0;i<NB_OA;i++)
625         for (j=0;j<NB_PREEMPTION;j++)
626         {
627             dblVecEnd(&(ls->bbw[i][j]));
628             dblVecEnd(&(ls->remoteBbw[i][j]));
629             dblVecEnd(&(ls->fbw[i][j]));

```

```

630         dblVecEnd(&(ls->remoteFbw[i][j]));
631     }
632     free(ls);
633
634     return 0;
635 }

```

4.11.3.22 int DBlinkStateEnd (DBLinkState * ls)

Definition at line 637 of file database-oli.c.

```

638 {
639     int i,j;
640
641     if (ls == NULL)
642     {
643         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
644             __FILE__, __LINE__);
645         return -1;
646     }
647
648     for (i=0;i<NB_OA;i++)
649         for (j=0;j<NB_PREEMPTION;j++)
650         {
651             dblVecEnd(&(ls->bbw[i][j]));
652             dblVecEnd(&(ls->remoteBbw[i][j]));
653             dblVecEnd(&(ls->fbw[i][j]));
654             dblVecEnd(&(ls->remoteFbw[i][j]));
655         }
656
657     return 0;
658 }

```

4.11.3.23 int DBlinkStateInit (DBLinkState * ls)

Definition at line 530 of file database-oli.c.

```

531 {
532     int i,j,k,l;
533
534     if (ls == NULL)
535     {
536         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
537             __FILE__, __LINE__);
538         return -1;
539     }
540
541     memset(ls, 0, sizeof(DBLinkState));
542
543     for (i=0;i<NB_OA;i++)
544         for (j=0;j<NB_PREEMPTION;j++)
545         {
546             if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
547             {
548                 for (k=i;k>=0;k++)
549                     for (l=j-1;l>=0;l++)
550                     {
551                         dblVecEnd(&(ls->bbw[k][l]));
552                         dblVecEnd(&(ls->remoteBbw[k][l]));
553                         dblVecEnd(&(ls->fbw[k][l]));
554                         dblVecEnd(&(ls->remoteFbw[k][l]));

```

```

555         }
556         addError(CRITICAL,"Unable to create link state in %s at line %d",
557                 __FILE__, __LINE__);
558         return -1;
559     }
560     else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
561     {
562         dblVecEnd(&(ls->bbw[i][j]));
563         for (k=i;k>=0;k++)
564             for (l=j-1;l>=0;l++)
565             {
566                 dblVecEnd(&(ls->bbw[k][l]));
567                 dblVecEnd(&(ls->remoteBbw[k][l]));
568                 dblVecEnd(&(ls->fbw[k][l]));
569                 dblVecEnd(&(ls->remoteFbw[k][l]));
570             }
571         addError(CRITICAL,"Unable to create link state in %s at line %d",
572                 __FILE__, __LINE__);
573         return -1;
574     }
575     else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
576     {
577         dblVecEnd(&(ls->bbw[i][j]));
578         dblVecEnd(&(ls->remoteBbw[i][j]));
579         for (k=i;k>=0;k++)
580             for (l=j-1;l>=0;l++)
581             {
582                 dblVecEnd(&(ls->bbw[k][l]));
583                 dblVecEnd(&(ls->remoteBbw[k][l]));
584                 dblVecEnd(&(ls->fbw[k][l]));
585                 dblVecEnd(&(ls->remoteFbw[k][l]));
586             }
587         addError(CRITICAL,"Unable to create link state in %s at line %d",
588                 __FILE__, __LINE__);
589         return -1;
590     }
591     else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
592     {
593         dblVecEnd(&(ls->bbw[i][j]));
594         dblVecEnd(&(ls->remoteBbw[i][j]));
595         dblVecEnd(&(ls->fbw[i][j]));
596         for (k=i;k>=0;k++)
597             for (l=j-1;l>=0;l++)
598             {
599                 dblVecEnd(&(ls->bbw[k][l]));
600                 dblVecEnd(&(ls->remoteBbw[k][l]));
601                 dblVecEnd(&(ls->fbw[k][l]));
602                 dblVecEnd(&(ls->remoteFbw[k][l]));
603             }
604         addError(CRITICAL,"Unable to create link state in %s at line %d",
605                 __FILE__, __LINE__);
606         return -1;
607     }
608     }
609     return 0;
610 }
611 }

```

4.11.3.24 DBLinkState* DBlinkStateNew ()

Definition at line 444 of file database-oli.c.

Referenced by computeBackup().

```
445 {
```

```

446 DBLinkState* ls;
447 int i,j,k,l;
448
449 if ((ls=calloc(1,sizeof(DBLinkState)))!=NULL)
450 {
451     addError(CRITICAL,"Critical lack of memory in %s at line %d",
452             __FILE__,__LINE__);
453     return NULL;
454 }
455
456 for (i=0;i<NB_OA;i++)
457     for (j=0;j<NB_PREEMPTION;j++)
458     {
459         if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
460         {
461             for (k=i;k>=0;k--)
462                 for (l=j-1;l>=0;l--)
463                 {
464                     dblVecEnd(&(ls->bbw[k][l]));
465                     dblVecEnd(&(ls->remoteBbw[k][l]));
466                     dblVecEnd(&(ls->fbw[k][l]));
467                     dblVecEnd(&(ls->remoteFbw[k][l]));
468                 }
469             free(ls);
470             addError(CRITICAL,"Unable to create link state in %s at line %d",
471                     __FILE__,__LINE__);
472             return NULL;
473         }
474         else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
475         {
476             dblVecEnd(&(ls->bbw[i][j]));
477             for (k=i;k>=0;k--)
478                 for (l=j-1;l>=0;l--)
479                 {
480                     dblVecEnd(&(ls->bbw[k][l]));
481                     dblVecEnd(&(ls->remoteBbw[k][l]));
482                     dblVecEnd(&(ls->fbw[k][l]));
483                     dblVecEnd(&(ls->remoteFbw[k][l]));
484                 }
485             free(ls);
486             addError(CRITICAL,"Unable to create link state in %s at line %d",
487                     __FILE__,__LINE__);
488             return NULL;
489         }
490         else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
491         {
492             dblVecEnd(&(ls->bbw[i][j]));
493             dblVecEnd(&(ls->remoteBbw[i][j]));
494             for (k=i;k>=0;k--)
495                 for (l=j-1;l>=0;l--)
496                 {
497                     dblVecEnd(&(ls->bbw[k][l]));
498                     dblVecEnd(&(ls->remoteBbw[k][l]));
499                     dblVecEnd(&(ls->fbw[k][l]));
500                     dblVecEnd(&(ls->remoteFbw[k][l]));
501                 }
502             free(ls);
503             addError(CRITICAL,"Unable to create link state in %s at line %d",
504                     __FILE__,__LINE__);
505             return NULL;
506         }
507         else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
508         {
509             dblVecEnd(&(ls->bbw[i][j]));
510             dblVecEnd(&(ls->remoteBbw[i][j]));
511             dblVecEnd(&(ls->fbw[i][j]));
512             for (k=i;k>=0;k--)

```

```

513         for (l=j-1;l>=0;l--)
514         {
515             dblVecEnd(&(ls->bbw[k][l]));
516             dblVecEnd(&(ls->remoteBbw[k][l]));
517             dblVecEnd(&(ls->fbw[k][l]));
518             dblVecEnd(&(ls->remoteFbw[k][l]));
519         }
520         free(ls);
521         addError(CRITICAL,"Unable to create link state in %s at line %d",
522             __FILE__,__LINE__);
523         return NULL;
524     }
525 }
526
527 return ls;
528 }

```

4.11.3.25 int DBLspCompare (const DBLabelSwitchedPath * LSPa, const DBLabelSwitchedPath * LSPb)

Definition at line 357 of file database-oli.c.

```

358 {
359     if (LSPa->precedence > LSPb->precedence)
360         return 1;
361     else if (LSPa->precedence < LSPb->precedence)
362         return -1;
363     else if (LSPa->bw[0] > LSPb->bw[0])
364         return 1;
365     else if (LSPa->bw[0] < LSPb->bw[0])
366         return -1;
367     else
368     {
369         if (LSPa->id < LSPb->id)
370             return 1;
371         else if (LSPa->id > LSPb->id)
372             return -1;
373     }
374
375     return 0;
376 }

```

4.11.3.26 int DBLspCopy (DBLabelSwitchedPath * dst, DBLabelSwitchedPath * src)

Definition at line 157 of file database-oli.c.

```

158 {
159     int ret=0;
160
161     if (dst == NULL || src==NULL)
162     {
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 }

```

4.11.3.27 int DBlspDestroy ([DBLabelSwitchedPath](#) * *lsp*)

Definition at line 122 of file database-oli.c.

```

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 }

```

4.11.3.28 int DBlspEnd ([DBLabelSwitchedPath](#) * *lsp*)

Definition at line 140 of file database-oli.c.

```

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 }

```

4.11.3.29 int DBlspInit (DBLabelSwitchedPath * lsp)

Definition at line 73 of file database-oli.c.

```

74 {
75     if (lsp == NULL)
76     {
77         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
78             __FILE__, __LINE__);
79         return -1;
80     }
81
82     if (longListInit(&(lsp->forbidLinks), -1) < 0)
83     {
84         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
85             __FILE__, __LINE__);
86         return -1;
87     }
88
89     if (longListInit(&(lsp->path), -1) < 0)
90     {
91         longListEnd(&(lsp->forbidLinks));
92         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
93             __FILE__, __LINE__);
94         return -1;
95     }
96
97     if (longListInit(&(lsp->primPath), -1) < 0)
98     {
99         longListEnd(&(lsp->path));
100         longListEnd(&(lsp->forbidLinks));
101         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
102             __FILE__, __LINE__);
103         return -1;
104     }
105
106     if (longListInit(&(lsp->backLSPIDs), -1) < 0)
107     {
108         longListEnd(&(lsp->primPath));
109         longListEnd(&(lsp->path));
110         longListEnd(&(lsp->forbidLinks));
111         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
112             __FILE__, __LINE__);
113         return -1;
114     }
115
116     memset(lsp->bw, 0, NB_OA * sizeof(double));
117     lsp->noContentionId = -1; //very important
118
119     return 0;
120 }

```

4.11.3.30 int DBlspListDestroy (DBLSPList * list)

Definition at line 251 of file database-oli.c.

```

252 {
253     if (list == NULL || list->cont == NULL)
254     {
255         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
256             __FILE__, __LINE__);
257         return -1;
258     }

```

```
259
260     free(list->cont);
261     free(list);
262
263     return 0;
264 }
```

4.11.3.31 int DBLspListEnd (**DBLSPList** * *list*)

Definition at line 266 of file database-oli.c.

```
267 {
268     if (list == NULL || list->cont == NULL)
269     {
270         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
271             __FILE__,__LINE__);
272         return -1;
273     }
274
275     free(list->cont);
276     list->cont = NULL;
277     list->size = 0;
278     list->top = 0;
279
280     return 0;
281 }
```

4.11.3.32 int DBLspListInit (**DBLSPList** * *list*, long *size*)

Definition at line 223 of file database-oli.c.

```
224 {
225     void* ptr=NULL;
226
227     if (list == NULL)
228     {
229         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
230             __FILE__,__LINE__);
231         return -1;
232     }
233
234     if (size == -1)
235         size = LSPLIST_INITSIZE;
236
237     if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
238     {
239         addError(CRITICAL,"Critical lack of memory in %s at line %d",
240             __FILE__,__LINE__);
241         return -1;
242     }
243
244     list->size = size;
245     list->top = 0;
246     list->cont = ptr;
247
248     return 0;
249 }
```


4.11.3.33 int DBlspListInsert (DBLSPList * list, DBLabelSwitchedPath * lsp)

Definition at line 283 of file database-oli.c.

```

284 {
285     int a,b;
286     void *ptr=NULL;
287
288     if (list == NULL || list->cont == NULL || lsp == NULL)
289     {
290         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
291             __FILE__,__LINE__);
292         return -1;
293     }
294
295     // check the capacity of the list
296     if (list->top >= list->size)
297     {
298         if ((ptr = realloc(list->cont, list->size
299             * 2 * sizeof(DBLabelSwitchedPath*))) == NULL)
300         {
301             addError(CRITICAL,"Critical lack of memory in %s at line %d",
302                 __FILE__,__LINE__);
303             return -1;
304         }
305         else
306         {
307             list->cont=ptr;
308             list->size*=2;
309         }
310     }
311
312     // find the position in the list (to keep it sorted)
313     a = 0;
314     b = list->top-1;
315
316     // empty list or after the last elem
317     if (list->top == 0 || DBlspCompare(list->cont[b], lsp) >= 0)
318     {
319         list->cont[list->top++] = lsp;
320         return (list->top-1);
321     }
322
323     // before the first elem
324     if (DBlspCompare(lsp, list->cont[a]) >= 0)
325     {
326         memmove(list->cont+1, list->cont, (list->top)*sizeof(void*));
327         list->cont[0] = lsp;
328         list->top++;
329         return 0;
330     }
331
332     // now the insert position is inside [a,b[
333     while (b - a > 1)
334     {
335         int mid = (a + b)/2;
336         int ret = DBlspCompare(lsp, list->cont[mid]);
337
338         if (ret == 1)
339             b = mid;
340         else if (ret == -1)
341             a = mid;
342         else // if (ret == 0)
343         {
344             a = mid;
345             b = mid;
346         }
347     }

```

```

347     }
348
349     // now insert before b
350     memmove(list->cont+b+1, list->cont+b, (list->top - b)*sizeof(void*));
351     list->cont[b] = lsp;
352     list->top++;
353
354     return b;
355 }

```

4.11.3.34 DBLSPList* DBlspListNew (long size)

Definition at line 193 of file database-oli.c.

```

194 {
195     DBLSPList *list=NULL;
196     void* ptr=NULL;
197
198     if ((list = calloc(1,sizeof(DBLSPList))) == NULL)
199     {
200         addError(CRITICAL,"Critical lack of memory in %s at line %d",
201             __FILE__,__LINE__);
202         return NULL;
203     }
204
205     if (size == -1)
206         size = LSPLIST_INITSIZE;
207
208     if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
209     {
210         addError(CRITICAL,"Critical lack of memory in %s at line %d",
211             __FILE__,__LINE__);
212         free(list);
213         return NULL;
214     }
215
216     list->size = size;
217     list->top = 0;
218     list->cont = ptr;
219
220     return list;
221 }

```

4.11.3.35 int DBlspListRemove (DBLSPList * list, DBLabelSwitchedPath * lsp)

Definition at line 378 of file database-oli.c.

```

379 {
380     int a,b,index;
381
382     if (list == NULL || list->cont == NULL || lsp == NULL)
383     {
384         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
385             __FILE__,__LINE__);
386         return -1;
387     }
388
389     // find the position in the list
390     a = 0;
391     b = list->top-1;
392

```

```

393     // empty list
394     if (list->top == 0)
395     {
396         addError(WARNING,"Removing inexistent LSP in %s at line %d",
397             __FILE__,__LINE__);
398         return -1;
399     }
400
401     while (b - a > 1)
402     {
403         int mid = (a + b)/2;
404         int ret = DBlspCompare(lsp, list->cont[mid]);
405
406         if (ret == 1)
407             b = mid;
408         else if (ret == -1)
409             a = mid;
410         else // if (ret == 0)
411         {
412             a = mid;
413             b = mid;
414         }
415     }
416
417     if (DBlspCompare(lsp, list->cont[a]) == 0)
418     {
419         index = a;
420     }
421     else if (DBlspCompare(lsp, list->cont[b]) == 0)
422     {
423         index = b;
424     }
425     else // not found
426     {
427         addError(WARNING,"Removing inexistent LSP in %s at line %d",
428             __FILE__,__LINE__);
429         return -1;
430     }
431
432     // now delete index
433     memmove(list->cont + index, list->cont + index + 1, (list->top - index - 1)*sizeof(void*));
434     list->top--;
435
436     return 0;
437 }

```

4.11.3.36 DBLabelSwitchedPath* DBlspNew ()

Definition at line 19 of file database-oli.c.

Referenced by DBaddLSP(), and evalLS().

```

20 {
21     DBLabelSwitchedPath* lsp;
22
23     if ((lsp=calloc(1,sizeof(DBLabelSwitchedPath)))==NULL)
24     {
25         addError(CRITICAL,"Critical lack of memory in %s at line %d",
26             __FILE__,__LINE__);
27         return NULL;
28     }
29
30     if (longListInit(&(lsp->forbidLinks),-1)<0)
31     {

```

```

32     free(lsp);
33     addError(CRITICAL,"Unable to create label switched path in %s at line %d",
34         __FILE__,__LINE__);
35     return NULL;
36 }
37
38 if (longListInit(&(lsp->path),-1)<0)
39 {
40     longListEnd(&(lsp->forbidLinks));
41     free(lsp);
42     addError(CRITICAL,"Unable to create label switched path in %s at line %d",
43         __FILE__,__LINE__);
44     return NULL;
45 }
46
47 if (longListInit(&(lsp->primPath),-1)<0)
48 {
49     longListEnd(&(lsp->path));
50     longListEnd(&(lsp->forbidLinks));
51     free(lsp);
52     addError(CRITICAL,"Unable to create label switched path in %s at line %d",
53         __FILE__,__LINE__);
54     return NULL;
55 }
56
57 if (longListInit(&(lsp->backLSPIDs),-1)<0)
58 {
59     longListEnd(&(lsp->primPath));
60     longListEnd(&(lsp->path));
61     longListEnd(&(lsp->forbidLinks));
62     free(lsp);
63     addError(CRITICAL,"Unable to create label switched path in %s at line %d",
64         __FILE__,__LINE__);
65     return NULL;
66 }
67
68 lsp->noContentionId=-1; //very important
69
70 return lsp;
71 }

```

4.11.3.37 DataBase* DBnew (long ID)

Definition at line 1280 of file database-oli.c.

```

1281 {
1282     DataBase *dataBase=NULL;
1283
1284     if ((dataBase=calloc(1,sizeof(DataBase)))==NULL)
1285     {
1286         addError(CRITICAL,"Critical lack of memory in %s at line %d",
1287             __FILE__,__LINE__);
1288         return NULL;
1289     }
1290
1291     dataBase->id=ID;
1292
1293     if (DBnodeVecInit(&(dataBase->nodeVec),-1)<0)
1294     {
1295         addError(CRITICAL,"Unable to initialize the general node container in %s at line %d",
1296             __FILE__,__LINE__);
1297         free(dataBase);
1298         return NULL;
1299     }

```

```

1300
1301     if (DBlspVecInit(&(dataBase->lspVec),-1)<0)
1302     {
1303         addError(CRITICAL,"Unable to initialize the general LSP container in %s at line %d",
1304             __FILE__,__LINE__);
1305         DBnodeVecEnd(&(dataBase->nodeVec));
1306         free(dataBase);
1307         return NULL;
1308     }
1309
1310     if (DBlinkTabInit(&(dataBase->linkTab),-1)<0)
1311     {
1312         addError(CRITICAL,"Unable to initialize the general link container in %s at line %d",
1313             __FILE__,__LINE__);
1314         DBnodeVecEnd(&(dataBase->nodeVec));
1315         DBlspVecEnd(&(dataBase->lspVec));
1316         free(dataBase);
1317         return NULL;
1318     }
1319
1320     if (longVecInit(&(dataBase->linkSrcVec),LINKTAB_INITSIZE)<0)
1321     {
1322         addError(CRITICAL,"Unable to initialize the link id-src translator in %s at line %d",
1323             __FILE__,__LINE__);
1324         DBnodeVecEnd(&(dataBase->nodeVec));
1325         DBlspVecEnd(&(dataBase->lspVec));
1326         DBlinkTabEnd(&(dataBase->linkTab));
1327         free(dataBase);
1328         return NULL;
1329     }
1330
1331     if (longVecInit(&(dataBase->linkDstVec),LINKTAB_INITSIZE)<0)
1332     {
1333         addError(CRITICAL,"Unable to initialize the link id-dst translator in %s at line %d",
1334             __FILE__,__LINE__);
1335         DBnodeVecEnd(&(dataBase->nodeVec));
1336         DBlspVecEnd(&(dataBase->lspVec));
1337         DBlinkTabEnd(&(dataBase->linkTab));
1338         longVecEnd(&(dataBase->linkSrcVec));
1339         free(dataBase);
1340         return NULL;
1341     }
1342
1343     dataBase->nbNodes=0;
1344     dataBase->nbLinks=0;
1345
1346     return dataBase;
1347 }

```

4.11.3.38 void DBprintDB (DataBase * db)

Definition at line 2253 of file database-oli.c.

```

2254 {
2255     long i,j;
2256
2257     printf("Printing info about nodes ...\n");
2258     printf("-----\n");
2259
2260     for (i=0; i<db->nodeVec.size; i++)
2261     {
2262         if (db->nodeVec.cont[i])
2263         {
2264             printf("Node id : %ld\n", i);

```

```

2265         printf("-----\n");
2266         DBprintNode(db->nodeVec.cont[i]);
2267     }
2268 }
2269
2270 printf("\nPrinting info about links ...\n");
2271 printf("-----\n");
2272
2273 for (i=0; i<db->linkTab.size; i++)
2274     for (j=0; j<db->linkTab.size; j++)
2275     {
2276         if (db->linkTab.cont[i][j])
2277         {
2278             printf("Link %ld-%ld (id = %ld)\n", i, j, DBgetLinkID(db, i, j));
2279             printf("-----\n");
2280
2281             DBprintLink(db->linkTab.cont[i][j]);
2282         }
2283     }
2284 }
2285 }

```

4.11.3.39 int DBremoveLink (DataBase * dataBase, long src, long dst)

Definition at line 1635 of file database-oli.c.

```

1636 {
1637     int id,ret=0;
1638
1639     if (dataBase == NULL)
1640     {
1641         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1642             __FILE__,__LINE__);
1643         return -1;
1644     }
1645
1646     if ((DBnodeVecGet(&(dataBase->nodeVec),src)==NULL) ||
1647         (DBnodeVecGet(&(dataBase->nodeVec),dst)==NULL) ||
1648         (DBlinkTabGet(&(dataBase->linkTab),src,dst)==NULL))
1649     {
1650         addError(CRITICAL,"Link doesn't exist or database unconsistency in %s at line %d",
1651             __FILE__,__LINE__);
1652         return -1;
1653     }
1654
1655     ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
1656     ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[dst]->inNeighb),src));
1657
1658     ANDERROR(ret,DBlinkTabRemove(&(dataBase->linkTab),src,dst));
1659
1660     id=DBgetLinkID(dataBase,src,dst);
1661     ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,0));
1662     ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,0));
1663
1664     while (dataBase->linkSrcVec.cont[dataBase->linkSrcVec.top-1] == 0)
1665         dataBase->linkSrcVec.top--;
1666
1667     if (ret<0)
1668     {
1669         addError(CRITICAL,"Link removal uncomplete in %s at line %d",
1670             __FILE__,__LINE__);
1671     }
1672
1673     dataBase->nbLinks--;

```

```

1674
1675     return ret;
1676 }

```

4.11.3.40 int DBremoveLSP (DataBase * dataBase, long id)

Definition at line 2005 of file database-oli.c.

```

2006 {
2007     DBLabelSwitchedPath *lsp=NULL, *contentLSP=NULL;
2008     int i,ret=0;
2009     DBLink *lnk=NULL;
2010     LongVec isProcessed;
2011
2012     if (dataBase == NULL)
2013     {
2014         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2015             __FILE__,__LINE__);
2016         return -1;
2017     }
2018
2019     if ((lsp = DBlspVecGet(&(dataBase->lspVec), id)) == NULL)
2020     {
2021         addError(CRITICAL,"Trying to remove inexistent LSP (id = %ld) in %s at line %d",
2022             id,__FILE__,__LINE__);
2023         return -1;
2024     }
2025
2026     if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
2027     {
2028         addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
2029             __FILE__,__LINE__);
2030         return -1;
2031     }
2032
2033     #if defined SIMULATOR
2034         // Remove the LSP from each link list and update all the linkstates
2035         for (i=0;i<lsp->path.top-1;i++)
2036         {
2037             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
2038                 lsp->path.cont[i+1]);
2039             ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
2040             ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->path.cont[i],
2041                 lsp->path.cont[i+1], &(lnk->state), lsp));
2042             isProcessed.cont[lnk->id] = 1;
2043         }
2044         if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
2045         {
2046             for (i=0;i<lsp->primPath.top-1;i++)
2047             {
2048                 lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
2049                     lsp->primPath.cont[i+1]);
2050                 if (isProcessed.cont[lnk->id] == 0)
2051                 {
2052                     ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->primPath.cont[i],
2053                         lsp->primPath.cont[i+1], &(lnk->state), lsp));
2054                     isProcessed.cont[lnk->id] = 1;
2055                 }
2056             }
2057         }
2058     #elif defined AGENT
2059         // Remove the LSP to the link attached to the agent and update the linkstate
2060         for (i=0;i<lsp->path.top-1;i++)
2061         {

```

```

2062         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
2063                         lsp->path.cont[i+1]);
2064         ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
2065
2066         if (lsp->path.cont[i] == dataBase->id)
2067         {
2068             ANDERROR(ret,DBupdateLSPOnRemove(dataBase, lsp->path.cont[i],
2069                                             lsp->path.cont[i+1], &(lnk->state), lsp));
2070             isProcessed.cont[lnk->id] = 1;
2071         }
2072     }
2073     if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
2074     {
2075         for (i=0;i<lsp->primPath.top-1;i++)
2076         {
2077             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
2078                             lsp->primPath.cont[i+1]);
2079
2080             if (lsp->primPath.cont[i] == dataBase->id)
2081             {
2082                 if (isProcessed.cont[lnk->id] == 0)
2083                 {
2084                     ANDERROR(ret,DBupdateLSPOnRemove(dataBase, lsp->primPath.cont[i],
2085                                                         lsp->primPath.cont[i+1], &(lnk->state), lsp));
2086                 }
2087                 break;
2088             }
2089         }
2090     }
2091     #else
2092         // Generate an error;
2093         COMPILER_ERROR;
2094     #endif
2095
2096     longVecEnd(&(isProcessed));
2097
2098     // remove the lsp from the global list
2099     ANDERROR(ret,DBlspVecRemove(&(dataBase->lspVec), id));
2100
2101     if (lsp->noContentionId>=0)
2102     {
2103         if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),lsp->noContentionId))==NULL)
2104         {
2105             addError(WARNING,"Unable to get no contention LSP in %s at line %d",
2106                     __FILE__,__LINE__);
2107             // not critical enough to abort
2108         }
2109         contentLSP->noContentionId=-1;
2110     }
2111
2112     // free the lsp
2113     DBlspDestroy(lsp);
2114
2115     if (ret<0)
2116     {
2117         addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
2118                 __FILE__,__LINE__);
2119     }
2120
2121     return ret;
2122 }

```

4.11.3.41 int DBremoveNode (DataBase * dataBase, long id)

Definition at line 1499 of file database-oli.c.


```

1500 {
1501     DBNode *node=NULL;
1502     int ret=0;
1503
1504     if (dataBase == NULL)
1505     {
1506         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1507             __FILE__,__LINE__);
1508         return -1;
1509     }
1510
1511     if ((node=DBnodeVecGet(&(dataBase->nodeVec),id)) == NULL)
1512     {
1513         addError(CRITICAL,"Trying to remove an inexistent node in %s at line %d",
1514             __FILE__,__LINE__);
1515         return -1;
1516     }
1517
1518     // remember that DBremoveLink will update the neighbour list
1519     while(node->inNeighb.top > 0)
1520     {
1521         ANDERROR(ret,DBremoveLink(dataBase,node->inNeighb.cont[node->inNeighb.top-1],id));
1522     }
1523
1524     // remember that DBremoveLink will update the neighbour list
1525     while(node->outNeighb.top > 0)
1526     {
1527         ANDERROR(ret,DBremoveLink(dataBase,id,node->outNeighb.cont[node->outNeighb.top-1]));
1528     }
1529
1530     ANDERROR(ret,DBnodeVecRemove(&(dataBase->nodeVec),id));
1531
1532     if (ret<0)
1533     {
1534         addError(CRITICAL,"Node removal uncomplete in %s at line %d",
1535             __FILE__,__LINE__);
1536     }
1537
1538     dataBase->nbLinks--;
1539
1540     return ret;
1541 }

```

4.11.3.42 int DBsetLinkState (**DataBase** * *dataBase*, long *src*, long *dst*, **DBLinkState** * *newLS*)

Definition at line 2180 of file database-oli.c.

```

2181 {
2182     DBLink *lnk=NULL;
2183
2184     if (dataBase == NULL || newLS == NULL)
2185     {
2186         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2187             __FILE__,__LINE__);
2188         return -1;
2189     }
2190
2191     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2192     {
2193         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2194             src,dst,__FILE__,__LINE__);
2195         return -1;
2196     }
2197

```

```

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, LSProuteInfo::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)

```

```

853     {
854         addError(CRITICAL,"Impossible to duplicate the linkState in %s at line %d",
855                 __FILE__,__LINE__);
856         return -1;
857     }
858
859     // now build a false LSP satisfying the request ....
860     lsp = DBlspNew();
861
862     lsp->precedence = req->precedence;
863     memcpy(lsp->bw, req->bw, NB_OA * sizeof(double));
864     longListCopy(&(lsp->forbidLinks), &(req->forbidLinks));
865
866     if (req->rerouteInfo.id >= 0)
867     {
868         lsp->noContentionId = req->rerouteInfo.id;
869     }
870
871     switch(req->type)
872     {
873         case PRIM:
874             lsp->type = PRIM;
875             lsp->primID = -1;
876             break;
877
878         case GLOBAL_BACK:
879         case LOCAL_BACK:
880             lsp->type = req->type;
881             lsp->primID = req->primID;
882
883             // look up the primary path ....
884             if ((primLSP = DBgetLSP(dataBase, lsp->primID)) == NULL)
885             {
886                 addError(CRITICAL,"Impossible to determine the primary path in %s at line %d",
887                         __FILE__,__LINE__);
888                 DBlspDestroy(lsp);
889                 return -1;
890             }
891
892             longListCopy(&(lsp->primPath), &(primLSP->path));
893
894             break;
895
896         default:
897             addError(CRITICAL,"Unknown request type (NULL) in %s at line %d",
898                     __FILE__,__LINE__);
899             DBlspDestroy(lsp);
900             return -1;
901     }
902
903     if (req->path.top < 2)
904     {
905         addError(CRITICAL,"Wrong path in request in %s at line %d",
906                 __FILE__,__LINE__);
907         DBlspDestroy(lsp);
908         return -1;
909     }
910
911     if (longListCopy(&(lsp->path), &(req->path)) < 0)
912     {
913         addError(CRITICAL,"Impossible to duplicate path in %s at line %d",
914                 __FILE__,__LINE__);
915         DBlspDestroy(lsp);
916         return -1;
917     }
918
919     ret = updateLS(dataBase, src, dst, newLS, lsp, op);

```

```

920
921     // clean up ....
922     DBlspDestroy(lsp);
923
924     return ret;
925 }

```

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.

```

928 {
929     bool path = FALSE;
930     int myPosPath=-1, myPosPrimPath=-1;
931     bool primPath = FALSE;
932     int i,oa;
933     int mult;
934     int plink;
935     double newBW[NB_OA];
936     DBLabelSwitchedPath* oldLSP=NULL;
937     bool rerouting = FALSE;
938
939     // check the arguments
940     if ((dataBase==NULL) || (ls==NULL) || (lsp==NULL))
941     {
942         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
943             __FILE__,__LINE__);
944         return -1;
945     }
946
947     // am i on the path ?
948     for (i=0; i<lsp->path.top; ++i)
949     {
950         if (lsp->path.cont[i] == src)
951             break;
952     }
953
954     if (i < (lsp->path.top - 1) && lsp->path.cont[i+1] == dst)
955     {
956         path = TRUE;
957         myPosPath = i;
958     }
959
960     // if i am on the path, am i the a rerouted LSP sharing the link with a preempted one ?
961     if (path == TRUE && lsp->noContentionId >= 0)
962     {
963         if ((oldLSP = DBgetLSP(dataBase, lsp->noContentionId)) == NULL)
964         {
965             addError(WARNING,"Cannot find the old LSP in %s at line %d",
966                 __FILE__,__LINE__);
967         }
968         else
969         {
970             for (i=0; i<oldLSP->path.top; ++i)
971             {

```

```

972             if (oldLSP->path.cont[i] == src)
973                 break;
974         }
975
976         if (i < (oldLSP->path.top - 1) && oldLSP->path.cont[i+1] == dst)
977         {
978             rerouting = TRUE;
979         }
980     }
981 }
982
983 // if rerouting -> check if there is a change in the bandwidth reservation. if no change we can le
984 if (rerouting == TRUE)
985 {
986     bool test=FALSE;
987
988     for (i=0; i<NB_OA; i++) {
989         if ((newBW[i] = max(lsp->bw[i] - oldLSP->bw[i], 0)) != 0)
990             test = TRUE;
991     }
992
993     if (test == FALSE)
994     {
995         return 0;
996     }
997 }
998
999 // if it is a backup am i on the prim path ?
1000 if (lsp->type == LOCAL_BACK || lsp->type == GLOBAL_BACK)
1001 {
1002     for (i=0; i<lsp->primPath.top; ++i)
1003     {
1004         if (lsp->primPath.cont[i] == src)
1005             break;
1006     }
1007
1008     if (i < (lsp->primPath.top - 1) && lsp->primPath.cont[i+1] == dst)
1009     {
1010         primPath = TRUE;
1011         myPosPrimPath = i;
1012     }
1013 }
1014
1015 if (!path && !primPath) // not concerned by this update ...
1016     return 0;
1017
1018 if (op == SETUP)
1019 {
1020     mult = 1;
1021 }
1022 else if (op == REMOVE)
1023 {
1024     mult = -1;
1025 }
1026 else
1027 {
1028     addError(CRITICAL, "Bad argument (unknown operation) in %s at line %d",
1029             __FILE__, __LINE__);
1030     return -1;
1031 }
1032
1033 if (lsp->type == LOCAL_BACK)
1034 {
1035     // the path is a local backup
1036     // -----
1037     int start, end;

```

```

1039
1040         // which link are we protecting ... and the start
1041         for (i=0; i<lsp->primPath.top; ++i)
1042         {
1043             if (lsp->primPath.cont[i] == lsp->path.cont[0])
1044                 break;
1045         }
1046
1047         if (i < lsp->primPath.top - 1)
1048         {
1049             plink = DBgetLinkID(dataBase, lsp->path.cont[0], lsp->primPath.cont[i+1]);
1050             start = i;
1051         }
1052         else
1053         {
1054             addError(CRITICAL, "Cannot determine the link to protect in %s at line %d",
1055                     __FILE__, __LINE__);
1056             return -1;
1057         }
1058
1059         if (path == TRUE)
1060         {
1061             // update bbw
1062             for (oa=0; oa<NB_OA; ++oa)
1063             {
1064                 if (ls->bbw[oa][lsp->precedence].size <= plink)
1065                     dblVecResize(&(ls->bbw[oa][lsp->precedence]), plink+1);
1066
1067                 if (rerouting == FALSE)
1068                     ls->bbw[oa][lsp->precedence].cont[plink] += (mult * lsp->bw[oa]);
1069                 else
1070                     ls->bbw[oa][lsp->precedence].cont[plink] += (mult * newBW[oa]);
1071             }
1072         }
1073
1074         if (primPath == TRUE)
1075         {
1076             // find the end
1077             for (i=start; i<lsp->primPath.top; ++i)
1078             {
1079                 if (lsp->primPath.cont[i] == lsp->path.cont[lsp->path.top - 1])
1080                     break;
1081             }
1082
1083             if (i < lsp->primPath.top)
1084             {
1085                 end = i;
1086             }
1087             else
1088             {
1089                 // finding an end is not required during path evaluation
1090                 end = -1;
1091                 /*
1092                 addError(CRITICAL, "Cannot determine the merging point in %s at line %d",
1093                         __FILE__, __LINE__);
1094                 return -1;
1095                 */
1096             }
1097
1098             // maybe fbw must be recomputed
1099             // if i'm before path.cont[0] or after path.cont[last]
1100             if (myPosPrimPath < start || myPosPrimPath >= end)
1101             {
1102                 for (oa=0; oa<NB_OA; ++oa)
1103                 {
1104                     if (ls->fbw[oa][lsp->precedence].size <= plink)
1105                         dblVecResize(&(ls->fbw[oa][lsp->precedence]), plink+1);

```

```

1106
1107         if (rerouting == FALSE)
1108             ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * lsp->bw[oa]);
1109         else
1110             ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * newBW[oa]);
1111     }
1112 }
1113 }
1114 }
1115 else if (lsp->type == GLOBAL_BACK)
1116 {
1117     // the path is a end-to-end backup
1118     // -----
1119     int start,end;
1120
1121     // we are protecting all nodes between path.cont[0] and path.cont[end]
1122     // find the start
1123     for (i=0; i<lsp->primPath.top; ++i)
1124     {
1125         if (lsp->primPath.cont[i] == lsp->path.cont[0])
1126             break;
1127     }
1128
1129     if (i < lsp->primPath.top)
1130     {
1131         start = i;
1132     }
1133     else
1134     {
1135         addError(CRITICAL,"Cannot determine the link to protect in %s at line %d",
1136             __FILE__,__LINE__);
1137         return -1;
1138     }
1139
1140     // find the end
1141     end = lsp->primPath.top - 1;
1142
1143     if (path == TRUE)
1144     {
1145         // update bbw
1146         for (i=start; i<end; ++i)
1147         {
1148             if ((plink = DBgetLinkID(dataBase, lsp->primPath.cont[i], lsp->primPath.cont[i+1])) >
1149                 for (oa=0; oa<NB_OA; ++oa)
1150                 {
1151                     if (ls->bbw[oa][lsp->precedence].size <= plink)
1152                         dblVecResize(&(ls->bbw[oa][lsp->precedence]), plink+1);
1153
1154                     if (rerouting == FALSE)
1155                         ls->bbw[oa][lsp->precedence].cont[plink] += (mult * lsp->bw[oa]);
1156                     else
1157                         ls->bbw[oa][lsp->precedence].cont[plink] += (mult * newBW[oa]);
1158                 }
1159             else
1160             {
1161                 addError(CRITICAL,"Cannot determine link ID in %s at line %d",
1162                     __FILE__,__LINE__);
1163                 return -1;
1164             }
1165         }
1166     }
1167
1168     if (primPath == TRUE)
1169     {
1170         // maybe fbw must be recomputed
1171         // if i'm before path.cont[0] or after path.cont[last]
1172

```

```

1173         for (i=0; i<start; ++i)
1174         {
1175             if ((plink = DBgetLinkID(dataBase, lsp->primPath.cont[i], lsp->primPath.cont[i+1])) > 0)
1176                 for (oa=0; oa<NB_OA; ++oa)
1177                 {
1178                     if (ls->fbw[oa][lsp->precedence].size <= plink)
1179                         dblVecResize(&(ls->fbw[oa][lsp->precedence]), plink+1);
1180
1181                     if (rerouting == FALSE)
1182                         ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * lsp->bw[oa]);
1183                     else
1184                         ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * newBW[oa]);
1185                 }
1186             else
1187             {
1188                 addError(CRITICAL, "Cannot determine link ID in %s at line %d",
1189                     __FILE__, __LINE__);
1190                 return -1;
1191             }
1192         }
1193
1194         for (i=end; i<lsp->primPath.top-1; ++i)
1195         {
1196             plink = DBgetLinkID(dataBase, lsp->primPath.cont[i], lsp->primPath.cont[i+1]);
1197             for (oa=0; oa<NB_OA; ++oa)
1198             {
1199                 if (ls->fbw[oa][lsp->precedence].size <= plink)
1200                     dblVecResize(&(ls->fbw[oa][lsp->precedence]), plink+1);
1201
1202                 if (rerouting == FALSE)
1203                     ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * lsp->bw[oa]);
1204                 else
1205                     ls->fbw[oa][lsp->precedence].cont[plink] -= (mult * newBW[oa]);
1206             }
1207         }
1208     }
1209 }
1210 else
1211 {
1212     // path is a primary
1213
1214     // update pbw
1215     for (i=0; i<NB_OA; ++i)
1216     {
1217         if (rerouting == FALSE)
1218             ls->pbw[i][lsp->precedence] += (mult * lsp->bw[i]);
1219         else
1220             ls->pbw[i][lsp->precedence] += (mult * newBW[i]);
1221     }
1222
1223     // update fbw
1224     for (i=0; i<lsp->path.top - 1; ++i)
1225     {
1226         int id;
1227
1228         if ((id = DBgetLinkID(dataBase, lsp->path.cont[i], lsp->path.cont[i+1])) >= 0)
1229             for (oa=0; oa<NB_OA; ++oa)
1230             {
1231                 if (ls->fbw[oa][lsp->precedence].size <= id)
1232                     dblVecResize(&(ls->fbw[oa][lsp->precedence]), id+1);
1233
1234                 if (rerouting == FALSE)
1235                     ls->fbw[oa][lsp->precedence].cont[id] += (mult * lsp->bw[oa]);
1236                 else
1237                     ls->fbw[oa][lsp->precedence].cont[id] += (mult * newBW[oa]);
1238             }
1239         else

```

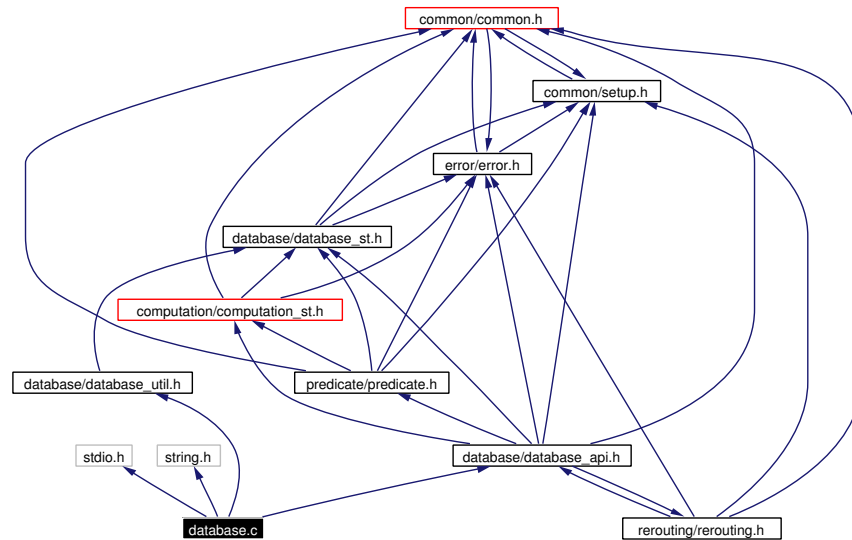


```
1240         {
1241             addError(CRITICAL,"Cannot determine link ID in %s at line %d",
1242                     __FILE__,__LINE__);
1243             return -1;
1244         }
1245     }
1246 }
1247
1248 computeRBW(dataBase, ls->rbw, ls->pbw, ls->bbw, ls->fbw);
1249
1250 return 0;
1251 }
```

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         {
```

```

741         if (dataBase->linkSrcVec.cont[i] == 0)
742             continue;
743         else
744             seenLinks++;
745
746         totBbw = 0;
747         totFbw = 0;
748
749         for(p=0; p<NB_PREEMPTION; ++p)
750         {
751             if (i < bbw[oa][p].size)
752                 totBbw += bbw[oa][p].cont[i];
753             if (i < fbw[oa][p].size)
754                 totFbw += fbw[oa][p].cont[i];
755
756             gbw[i].cont[p] = max(0, totBbw - totFbw);
757         }
758     }
759
760     seenNodes = 0;
761
762     // phase 1b (nodes or any set of links)
763     for (i=0; seenNodes<dataBase->nbNodes; ++i)
764     {
765         if (dataBase->nodeVec.cont[i] == NULL)
766             continue;
767         else
768             seenNodes++;
769
770         totBbw = 0;
771         totFbw = 0;
772
773         for(p=0; p<NB_PREEMPTION; ++p)
774         {
775             LongList* lst;
776             if ((lst = DBgetNodeInNeighb(dataBase, i)) != NULL)
777             {
778                 int l;
779                 for (l=0; l<lst->top; ++l)
780                 {
781                     int lnkID = DBgetLinkID(dataBase, lst->cont[l], i);
782                     if (lnkID < bbw[oa][p].size)
783                         totBbw += bbw[oa][p].cont[lnkID];
784                     if (lnkID < fbw[oa][p].size)
785                         totFbw += fbw[oa][p].cont[lnkID];
786                 }
787             }
788
789             gbw[i + nbLink].cont[p] = max(0, totBbw - totFbw);
790         }
791     }
792
793     // phase 2
794     oldM = 0;
795
796     for (p=0; p<NB_PREEMPTION; ++p)
797     {
798         m = 0;
799         for (i=0; i<nbLink + nbNode; ++i)
800         {
801             if (gbw[i].cont[p] > m)
802             {
803                 m = gbw[i].cont[p];
804             }
805         }
806
807         rbw[oa][p] = pbw[oa][p] + m - oldM;

```

```

808         oldM = m;
809     }
810 }
811
812 for (i=0; i<nbLink + nbNode; ++i)
813     dblVecEnd(&(gbw[i]));
814
815 free(gbw);
816 */
817
818 for (oa=0; oa<NB_OA; ++oa)
819     for (p=0; p<NB_PREEMPTION; ++p)
820     {
821         rbw[oa][p] = pbw[oa][p];
822     }
823
824 #if defined LINUX && defined TIME1
825     gettimeofday(&t2, &tz);
826     fprintf(stderr, "Time to compute rbw : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
827         (t2.tv_usec - t1.tv_usec) / 1000.0);
828 #endif
829
830 return 0;
831
832 }

```

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.

```

1613 {
1614     DBLink* link=NULL;
1615     int ret=0;
1616
1617     if (dataBase == NULL || initLinkState==NULL
1618         || id < 0 || src < 0 || dst < 0)
1619     {
1620         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
1621             __FILE__, __LINE__);
1622         return -1;
1623     }
1624
1625     if (((id < dataBase->linkSrcVec.size) && (dataBase->linkSrcVec.cont[id] > 0))
1626         ||
1627         ((id < dataBase->linkDstVec.size) && (dataBase->linkDstVec.cont[id] > 0)))
1628     {
1629         addError(CRITICAL, "Trying to add a link with a reserved ID (ID=%ld) in %s at line %d",
1630             id, __FILE__, __LINE__);
1631         return -1;
1632     }
1633
1634     if ((link = DBlinkNew()) == NULL)
1635     {
1636         addError(CRITICAL, "Unable to create link in %s at line %d",
1637             __FILE__, __LINE__);
1638         return -1;

```

```

1639     }
1640
1641     link->id=id;
1642
1643     if (DBlinkStateCopy(&(link->state), initLinkState))
1644     {
1645         addError(CRITICAL,"Unable to create link in %s at line %d",
1646                 __FILE__,__LINE__);
1647         DBlinkDestroy(link);
1648         return -1;
1649     }
1650
1651     if ((DBnodeVecGet(&(dataBase->nodeVec),src) == NULL) ||
1652         (DBnodeVecGet(&(dataBase->nodeVec),dst) == NULL))
1653     {
1654         addError(CRITICAL,"Source or destination node doesn't exist in %s at line %d",
1655                 __FILE__,__LINE__);
1656         DBlinkDestroy(link);
1657         return -1;
1658     }
1659
1660     if (DBlinkTabSet(&(dataBase->linkTab),link,src,dst)<0)
1661     {
1662         addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %d",
1663                 __FILE__,__LINE__);
1664         DBlinkDestroy(link);
1665         return -1;
1666     }
1667
1668     ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
1669     ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[dst]->inNeighb),src));
1670
1671     ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[src]->outNeighb)));
1672     ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[dst]->inNeighb)));
1673
1674     ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,src+1));
1675     ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,dst+1));
1676
1677     // Maximum non-null element
1678     dataBase->linkSrcVec.top = max(dataBase->linkSrcVec.top, id+1);
1679     dataBase->linkDstVec.top = dataBase->linkSrcVec.top;
1680
1681     if (ret<0)
1682     {
1683         addError(CRITICAL,"Link addition uncomplete in %s at line %d",
1684                 __FILE__,__LINE__);
1685     }
1686
1687     dataBase->nbLinks++;
1688
1689     return ret;
1690 }

```

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(), DBupdateLSPOnSetup(), 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.

```

1737 {
1738     DBLabelSwitchedPath *newLSP, *contentLSP=NULL;
1739     DBLSPList *lspList;
1740     int i,ret=0;
1741     DBLink *lnk=NULL;
1742     LongVec isProcessed;
1743     double rerouteGain[NB_OA];
1744     bool allowLSP=TRUE;
1745 #if defined SIMULATOR
1746     LongList idList;
1747 #elif defined AGENT
1748     int j;
1749     bool inPath=FALSE;
1750 #endif
1751
1752 #if defined LINUX && defined TIME2
1753     struct timezone tz;
1754     struct timeval t1,t2;
1755 #endif
1756
1757     if (dataBase == NULL || lsp==NULL)
1758     {
1759         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1760             __FILE__,__LINE__);
1761         return -1;
1762     }
1763
1764 #if defined LINUX && defined TIME2
1765     gettimeofday(&t1, &tz);
1766 #endif
1767
1768     if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
1769     {
1770         addError(CRITICAL,"Unable to initialize vector of longs in %s at line %d",
1771             __FILE__,__LINE__);
1772         return -1;
1773     }
1774
1775     memset(rerouteGain,0,NB_OA*sizeof(double));
1776
1777     // Check if establishment is possible
1778 #if defined SIMULATOR
1779     if (longListInit(&(idList),-1)<0)
1780     {
1781         addError(CRITICAL,"Unable to initialize list of longs in %s at line %d",
1782             __FILE__,__LINE__);
1783         return -1;
1784     }
1785     for (i=0;(i<lsp->path.top-1) && allowLSP;i++)
1786     {
1787         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
1788             lsp->path.cont[i+1]);
1789
1790
1791
1792         allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
1793             &(lnk->state),lsp,rerouteGain);
1794         if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
1795         {
1796             addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1797                 __FILE__,__LINE__);
1798             longListEnd(&(idList));
1799             longVecEnd(&(isProcessed));
1800             return -1;

```



```

1801     }
1802     idList.top=0;
1803     if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
1804     {
1805         addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1806             __FILE__,__LINE__);
1807         longListEnd(&(idList));
1808         longVecEnd(&(isProcessed));
1809         return -1;
1810     }
1811     if (longListMerge(&(idList),preemptList,preemptList)<0)
1812     {
1813         addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
1814             __FILE__,__LINE__);
1815         longListEnd(&(idList));
1816         longVecEnd(&(isProcessed));
1817         return -1;
1818     }
1819     isProcessed.cont[lnk->id] = 1;
1820 }
1821 if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
1822 {
1823     for (i=0;(i<lsp->primPath.top-1) && allowLSP;i++)
1824     {
1825         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
1826             lsp->primPath.cont[i+1]);
1827         if (isProcessed.cont[lnk->id] == 0)
1828         {
1829             allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1],
1830                 &(lnk->state),lsp,rerouteGain);
1831             if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==0)
1832             {
1833                 addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1834                     __FILE__,__LINE__);
1835                 longListEnd(&(idList));
1836                 longVecEnd(&(isProcessed));
1837                 return -1;
1838             }
1839             idList.top=0;
1840             if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
1841             {
1842                 addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1843                     __FILE__,__LINE__);
1844                 longListEnd(&(idList));
1845                 longVecEnd(&(isProcessed));
1846                 return -1;
1847             }
1848             if (longListMerge(&(idList),preemptList,preemptList)<0)
1849             {
1850                 addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
1851                     __FILE__,__LINE__);
1852                 longListEnd(&(idList));
1853                 longVecEnd(&(isProcessed));
1854                 return -1;
1855             }
1856             isProcessed.cont[lnk->id] = 1;
1857         }
1858     }
1859 }
1860 longListEnd(&(idList));
1861 #elif defined AGENT
1862     for (i=0;(i<lsp->path.top-1) && (lsp->path.cont[i]!=dataBase->id);i++);
1863
1864     if (i<lsp->path.top-1)
1865     {
1866         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
1867             lsp->path.cont[i+1]);

```

```

1868         allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
1869                                             &(lnk->state),lsp,rerouteGain);
1870     if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
1871     {
1872         addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1873                 __FILE__,__LINE__);
1874         longVecEnd(&(isProcessed));
1875     }
1876     if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)<0)
1877     {
1878         addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1879                 __FILE__,__LINE__);
1880         longVecEnd(&(isProcessed));
1881         return -1;
1882     }
1883     isProcessed.cont[lnk->id] = 1;
1884     inPath=TRUE;
1885 }
1886 if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
1887 {
1888     for (j=0;(j<lsp->primPath.top-1) && (lsp->primPath.cont[j]!=dataBase->id);j++);
1889
1890     if (j<lsp->primPath.top-1)
1891     {
1892         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[j],
1893                         lsp->primPath.cont[j+1]);
1894         if (isProcessed.cont[lnk->id] == 0)
1895         {
1896             allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[j],lsp->primPath.cont[j+1],
1897                                                 &(lnk->state),lsp,rerouteGain);
1898             if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[j],lsp->primPath.cont[j+1]))==NULL)
1899             {
1900                 addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1901                         __FILE__,__LINE__);
1902                 longVecEnd(&(isProcessed));
1903             }
1904             if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,preemptList)<0)
1905             {
1906                 addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1907                         __FILE__,__LINE__);
1908                 longVecEnd(&(isProcessed));
1909                 return -1;
1910             }
1911             isProcessed.cont[lnk->id] = 1;
1912         }
1913         inPath=TRUE;
1914     }
1915 }
1916 if (!inPath)
1917 {
1918     addError(CRITICAL,"Agent not concerned by this LSP in %s at line %d",
1919             __FILE__,__LINE__);
1920     longVecEnd(&(isProcessed));
1921     return -1;
1922 }
1923 #else
1924     // Generate an error;
1925     COMPILER_ERROR;
1926 #endif
1927
1928 if (!allowLSP)
1929 {
1930     addError(CRITICAL,"LSP refused by the predicate in %s at line %d",
1931             __FILE__,__LINE__);
1932     longVecEnd(&(isProcessed));
1933     return -1;
1934 }

```

```

1935
1936
1937     if ((newLSP=DBlspNew())==NULL)
1938     {
1939         addError(CRITICAL,"Unable to create LSP in %s at line %d",
1940             __FILE__,__LINE__);
1941         longVecEnd(&(isProcessed));
1942         return -1;
1943     }
1944
1945     if (DBlspCopy(newLSP,lsp)<0)
1946     {
1947         addError(CRITICAL,"Unable to create a valid LSP copy in %s at line %d",
1948             __FILE__,__LINE__);
1949         DBlspDestroy(newLSP);
1950         longVecEnd(&(isProcessed));
1951         return -1;
1952     }
1953
1954     if (DBlspVecSet(&(dataBase->lspVec),newLSP,newLSP->id)<0)
1955     {
1956         addError(CRITICAL,"Unable to insert LSP in the general LSP container in %s at line %d",
1957             __FILE__,__LINE__);
1958         DBlspDestroy(newLSP);
1959         longVecEnd(&(isProcessed));
1960         return -1;
1961     }
1962
1963     if (newLSP->noContentionId>=0)
1964     {
1965         if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),newLSP->noContentionId))==NULL)
1966         {
1967             addError(WARNING,"Unable to get no contention LSP in %s at line %d",
1968                 __FILE__,__LINE__);
1969             newLSP->noContentionId=-1;
1970             // not critical enough to abort
1971         }
1972         else
1973         {
1974             contentLSP->noContentionId=newLSP->id;
1975         }
1976     }
1977
1978     for (i=0;i<isProcessed.size;i++)
1979     {
1980         isProcessed.cont[i]=0;
1981     }
1982
1983
1984 #if defined SIMULATOR
1985     // Add the LSP to each link list and update all the linkstates (only once !!!!!)
1986     for (i=0;i<newLSP->path.top-1;i++)
1987     {
1988         lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
1989             newLSP->path.cont[i+1]);
1990         ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
1991         ANDERROR(ret,DBupdateLSPonSetup(dataBase, newLSP->path.cont[i],
1992             newLSP->path.cont[i+1], &(lnk->state), newLSP));
1993         isProcessed.cont[lnk->id] = 1;
1994     }
1995     if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
1996     {
1997         for (i=0;i<newLSP->primPath.top-1;i++)
1998         {
1999             lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
2000                 newLSP->primPath.cont[i+1]);
2001             if (isProcessed.cont[lnk->id] == 0)

```

```

2002         {
2003             ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->primPath.cont[i],
2004                                             newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
2005             isProcessed.cont[lnk->id] = 1;
2006         }
2007     }
2008 }
2009 #elif defined AGENT
2010 // Add the LSP to the link attached to the agent and update the linkstate
2011 for (i=0;i<newLSP->path.top-1;i++)
2012 {
2013     lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
2014                     newLSP->path.cont[i+1]);
2015     ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
2016
2017     if (newLSP->path.cont[i] == dataBase->id)
2018     {
2019         ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->path.cont[i],
2020                                         newLSP->path.cont[i+1], &(lnk->state), newLSP));
2021         isProcessed.cont[lnk->id] = 1;
2022     }
2023 }
2024 if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
2025 {
2026     for (i=0;i<newLSP->primPath.top-1;i++)
2027     {
2028         lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
2029                         newLSP->primPath.cont[i+1]);
2030
2031         if (newLSP->primPath.cont[i] == dataBase->id)
2032         {
2033             if (isProcessed.cont[lnk->id] == 0)
2034             {
2035                 ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->primPath.cont[i],
2036                                                 newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
2037             }
2038             break;
2039         }
2040     }
2041 }
2042 #else
2043 // Generate an error;
2044 COMPILER_ERROR;
2045 #endif
2046
2047 longVecEnd(&(isProcessed));
2048
2049 #if defined LINUX && defined TIME2
2050 gettimeofday(&t2, &tz);
2051 fprintf(stderr, "Time to add a new LSP : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
2052          (t2.tv_usec - t1.tv_usec) / 1000.0);
2053 #endif
2054
2055
2056 if (ret<0)
2057 {
2058     addError(CRITICAL,"LSP addition uncomplete in %s at line %d",
2059             __FILE__, __LINE__);
2060 }
2061
2062 return ret;
2063 }

```

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`.

```

1524 {
1525     DBNode *node=NULL;
1526
1527     if (dataBase == NULL)
1528     {
1529         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1530             __FILE__,__LINE__);
1531         return -1;
1532     }
1533
1534     if ((node=DBnodeNew()) == NULL)
1535     {
1536         addError(CRITICAL,"Unable to create node in %s at line %d",
1537             __FILE__,__LINE__);
1538         return -1;
1539     }
1540
1541     node->id=id;
1542
1543     if (DBnodeVecSet(&(dataBase->nodeVec),node,id) < 0)
1544     {
1545         addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %d",
1546             __FILE__,__LINE__);
1547         DBnodeDestroy(node);
1548         return -1;
1549     }
1550
1551     dataBase->nbNodes++;
1552
1553     return 0;
1554 }
```

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`.

```

1407 {
1408     if (dataBase == NULL)
1409     {
1410         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1411             __FILE__,__LINE__);
1412         return -1;
1413     }
1414
1415     DBnodeVecEnd(&(dataBase->nodeVec));
1416     DBlspVecEnd(&(dataBase->lspVec));
1417     DBlinkTabEnd(&(dataBase->linkTab));
1418     longVecEnd(&(dataBase->linkSrcVec));
1419     longVecEnd(&(dataBase->linkDstVec));
1420
1421     free(dataBase);
1422 }
```

```

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().

```

1479 {
1480     long ret;
1481
1482     if (dataBase == NULL)
1483     {
1484         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1485             __FILE__,__LINE__);
1486         return -1;
1487     }
1488
1489     if (longVecGet(&(dataBase->linkDstVec),id,&ret)<0)
1490     {
1491         addError(CRITICAL,"Inexistent link in %s at line %d",
1492             __FILE__,__LINE__);
1493         return -1;
1494     }
1495
1496     return (ret-1);
1497 }

```

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()`.

```

1439 {
1440     DBLink *lnk=NULL;
1441
1442     if (dataBase == NULL || src < 0 || dst < 0)
1443     {
1444         addError(CRITICAL,"Bad argument (NULL or negative value) in %s at line %d",
1445             __FILE__,__LINE__);
1446         return -1;
1447     }
1448
1449     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst))==NULL)
1450     {
1451         return -1;
1452     }
1453
1454     return lnk->id;
1455 }

```

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()`.

```

2199 {
2200     DBLink *lnk=NULL;
2201
2202     if (dataBase == NULL)
2203     {
2204         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2205             __FILE__,__LINE__);

```

```

2206         return NULL;
2207     }
2208
2209     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2210     {
2211         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2212                 src,dst,__FILE__,__LINE__);
2213         return NULL;
2214     }
2215
2216     return &(lnk->lspList);
2217 }

```

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().

```

1458 {
1459     long ret;
1460
1461     if (dataBase == NULL)
1462     {
1463         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1464                 __FILE__,__LINE__);
1465         return -1;
1466     }
1467
1468     if (longVecGet(&(dataBase->linkSrcVec),id,&ret)<0)
1469     {
1470         addError(CRITICAL,"Inexistent link in %s at line %d",
1471                 __FILE__,__LINE__);
1472         return -1;
1473     }
1474
1475     return (ret-1);
1476 }

```

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().

```

2220 {
2221     DBLink *lnk=NULL;
2222
2223     if (dataBase == NULL)
2224     {
2225         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2226                 __FILE__,__LINE__);
2227         return NULL;
2228     }
2229
2230     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2231     {

```



```

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",

```

```

1605         __FILE__, __LINE__);
1606     return -1;
1607 }
1608
1609     return dataBase->nbLinks;
1610 }

```

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().

```

1500 {
1501     if (dataBase == NULL)
1502     {
1503         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
1504             __FILE__, __LINE__);
1505         return -1;
1506     }
1507
1508     return dataBase->nbNodes;
1509 }

```

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().

```

2270 {
2271     DBNode *node=NULL;
2272
2273     if (dataBase == NULL)
2274     {
2275         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
2276             __FILE__, __LINE__);
2277         return NULL;
2278     }
2279
2280     if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
2281     {
2282         addError(CRITICAL, "Node don't exist (id = %ld) in %s at line %d",
2283             id, __FILE__, __LINE__);
2284         return NULL;
2285     }
2286
2287     return (&(node->inNeighb));
2288 }

```

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().

```

2292 {
2293     DBNode *node=NULL;
2294
2295     if (dataBase == NULL)
2296     {
2297         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2298             __FILE__,__LINE__);
2299         return NULL;
2300     }
2301
2302     if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
2303     {
2304         addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
2305             id,__FILE__,__LINE__);
2306         return NULL;
2307     }
2308
2309     return (&(node->outNeighb));
2310 }

```

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().

```

661 {
662     int i,j,ret=0;
663
664     if (dst == NULL || src == NULL)
665     {
666         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
667             __FILE__,__LINE__);
668         return -1;
669     }
670
671     dst->color=src->color;
672     memcpy(&(dst->cap),&(src->cap),NB_OA * sizeof(double));
673     memcpy(&(dst->rbw),&(src->rbw),NB_OA * NB_PREEMPTION * sizeof(double));
674     memcpy(&(dst->pbw),&(src->pbw),NB_OA * NB_PREEMPTION * sizeof(double));
675
676     for (i=0;(i<NB_OA && ret>=0);i++)
677         for (j=0;(j<NB_PREEMPTION && ret>=0);j++)
678         {
679             ANDERROR(ret,dblVecCopy(&(dst->bbw[i][j]),&(src->bbw[i][j])));
680             ANDERROR(ret,dblVecCopy(&(dst->remoteBbw[i][j]),&(src->remoteBbw[i][j])));
681             ANDERROR(ret,dblVecCopy(&(dst->fbw[i][j]),&(src->fbw[i][j])));
682             ANDERROR(ret,dblVecCopy(&(dst->remoteFbw[i][j]),&(src->remoteFbw[i][j])));
683         }
684
685     if (ret<0)
686     {
687         addError(CRITICAL,"Link state copy uncomplete in %s at line %d",
688             __FILE__,__LINE__);
689     }
690
691     return ret;
692 }

```

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().

```

614 {
615     int i,j;
616
617     if (ls == NULL)
618     {
619         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
620             __FILE__,__LINE__);
621         return -1;
622     }
623
624     for (i=0;i<NB_OA;i++)
625         for (j=0;j<NB_PREEMPTION;j++)
626         {
627             dblVecEnd(&(ls->bbw[i][j]));
628             dblVecEnd(&(ls->remoteBbw[i][j]));
629             dblVecEnd(&(ls->fbw[i][j]));
630             dblVecEnd(&(ls->remoteFbw[i][j]));
631         }
632     free(ls);
633
634     return 0;
635 }
```

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().

```

638 {
639     int i,j;
640
641     if (ls == NULL)
642     {
643         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
644             __FILE__,__LINE__);
645         return -1;
646     }
647
648     for (i=0;i<NB_OA;i++)
649         for (j=0;j<NB_PREEMPTION;j++)
650         {
651             dblVecEnd(&(ls->bbw[i][j]));
652             dblVecEnd(&(ls->remoteBbw[i][j]));
653             dblVecEnd(&(ls->fbw[i][j]));
654             dblVecEnd(&(ls->remoteFbw[i][j]));
655         }
656
657     return 0;
658 }
```

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().

```

531 {
532     int i,j,k,l;
533
534     if (ls == NULL)
535     {
536         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
537                 __FILE__,__LINE__);
538         return -1;
539     }
540
541     memset(ls, 0, sizeof(DBLinkState));
542
543     for (i=0;i<NB_OA;i++)
544         for (j=0;j<NB_PREEMPTION;j++)
545         {
546             if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
547             {
548                 for (k=i;k>=0;k++)
549                     for (l=j-1;l>=0;l++)
550                     {
551                         dblVecEnd(&(ls->bbw[k][l]));
552                         dblVecEnd(&(ls->remoteBbw[k][l]));
553                         dblVecEnd(&(ls->fbw[k][l]));
554                         dblVecEnd(&(ls->remoteFbw[k][l]));
555                     }
556                 addError(CRITICAL,"Unable to create link state in %s at line %d",
557                         __FILE__,__LINE__);
558                 return -1;
559             }
560             else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
561             {
562                 dblVecEnd(&(ls->bbw[i][j]));
563                 for (k=i;k>=0;k++)
564                     for (l=j-1;l>=0;l++)
565                     {
566                         dblVecEnd(&(ls->bbw[k][l]));
567                         dblVecEnd(&(ls->remoteBbw[k][l]));
568                         dblVecEnd(&(ls->fbw[k][l]));
569                         dblVecEnd(&(ls->remoteFbw[k][l]));
570                     }
571                 addError(CRITICAL,"Unable to create link state in %s at line %d",
572                         __FILE__,__LINE__);
573                 return -1;
574             }
575             else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
576             {
577                 dblVecEnd(&(ls->bbw[i][j]));
578                 dblVecEnd(&(ls->remoteBbw[i][j]));
579                 for (k=i;k>=0;k++)
580                     for (l=j-1;l>=0;l++)
581                     {
582                         dblVecEnd(&(ls->bbw[k][l]));
583                         dblVecEnd(&(ls->remoteBbw[k][l]));
584                         dblVecEnd(&(ls->fbw[k][l]));
585                         dblVecEnd(&(ls->remoteFbw[k][l]));
586                     }
587                 addError(CRITICAL,"Unable to create link state in %s at line %d",
588                         __FILE__,__LINE__);

```

```

589         return -1;
590     }
591     else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
592     {
593         dblVecEnd(&(ls->bbw[i][j]));
594         dblVecEnd(&(ls->remoteBbw[i][j]));
595         dblVecEnd(&(ls->fbw[i][j]));
596         for (k=i;k>=0;k++)
597             for (l=j-1;l>=0;l++)
598             {
599                 dblVecEnd(&(ls->bbw[k][l]));
600                 dblVecEnd(&(ls->remoteBbw[k][l]));
601                 dblVecEnd(&(ls->fbw[k][l]));
602                 dblVecEnd(&(ls->remoteFbw[k][l]));
603             }
604         addError(CRITICAL,"Unable to create link state in %s at line %d",
605                 __FILE__,__LINE__);
606         return -1;
607     }
608 }
609
610 return 0;
611 }

```

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.

```

445 {
446     DBLinkState* ls;
447     int i,j,k,l;
448
449     if ((ls=calloc(1,sizeof(DBLinkState)))==NULL)
450     {
451         addError(CRITICAL,"Critical lack of memory in %s at line %d",
452                 __FILE__,__LINE__);
453         return NULL;
454     }
455
456     for (i=0;i<NB_OA;i++)
457         for (j=0;j<NB_PREEMPTION;j++)
458         {
459             if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
460             {
461                 for (k=i;k>=0;k--)
462                     for (l=j-1;l>=0;l--)
463                     {
464                         dblVecEnd(&(ls->bbw[k][l]));
465                         dblVecEnd(&(ls->remoteBbw[k][l]));
466                         dblVecEnd(&(ls->fbw[k][l]));
467                         dblVecEnd(&(ls->remoteFbw[k][l]));
468                     }
469                 free(ls);
470                 addError(CRITICAL,"Unable to create link state in %s at line %d",
471                         __FILE__,__LINE__);
472                 return NULL;
473             }
474             else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
475             {
476                 dblVecEnd(&(ls->bbw[i][j]));
477                 for (k=i;k>=0;k--)

```

```

478         for (l=j-1;l>=0;l--)
479         {
480             dblVecEnd(&(ls->bbw[k][l]));
481             dblVecEnd(&(ls->remoteBbw[k][l]));
482             dblVecEnd(&(ls->fbw[k][l]));
483             dblVecEnd(&(ls->remoteFbw[k][l]));
484         }
485         free(ls);
486         addError(CRITICAL,"Unable to create link state in %s at line %d",
487             __FILE__, __LINE__);
488         return NULL;
489     }
490     else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
491     {
492         dblVecEnd(&(ls->bbw[i][j]));
493         dblVecEnd(&(ls->remoteBbw[i][j]));
494         for (k=i;k>=0;k--)
495             for (l=j-1;l>=0;l--)
496             {
497                 dblVecEnd(&(ls->bbw[k][l]));
498                 dblVecEnd(&(ls->remoteBbw[k][l]));
499                 dblVecEnd(&(ls->fbw[k][l]));
500                 dblVecEnd(&(ls->remoteFbw[k][l]));
501             }
502         free(ls);
503         addError(CRITICAL,"Unable to create link state in %s at line %d",
504             __FILE__, __LINE__);
505         return NULL;
506     }
507     else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
508     {
509         dblVecEnd(&(ls->bbw[i][j]));
510         dblVecEnd(&(ls->remoteBbw[i][j]));
511         dblVecEnd(&(ls->fbw[i][j]));
512         for (k=i;k>=0;k--)
513             for (l=j-1;l>=0;l--)
514             {
515                 dblVecEnd(&(ls->bbw[k][l]));
516                 dblVecEnd(&(ls->remoteBbw[k][l]));
517                 dblVecEnd(&(ls->fbw[k][l]));
518                 dblVecEnd(&(ls->remoteFbw[k][l]));
519             }
520         free(ls);
521         addError(CRITICAL,"Unable to create link state in %s at line %d",
522             __FILE__, __LINE__);
523         return NULL;
524     }
525 }
526
527 return ls;
528 }

```

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().

```

358 {
359     if (LSPa->precedence > LSPb->precedence)

```

```

360     return 1;
361     else if (LSPa->precedence < LSPb->precedence)
362         return -1;
363     else if (LSPa->bw[0] > LSPb->bw[0])
364         return 1;
365     else if (LSPa->bw[0] < LSPb->bw[0])
366         return -1;
367     else
368     {
369         if (LSPa->id < LSPb->id)
370             return 1;
371         else if (LSPa->id > LSPb->id)
372             return -1;
373     }
374
375     return 0;
376 }

```

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().

```

158 {
159     int ret=0;
160
161     if (dst == NULL || src==NULL)
162     {
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 }

```


4.12.3.27 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)
```

```

76     {
77         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
78             __FILE__,__LINE__);
79         return -1;
80     }
81
82     if (longListInit(&(lsp->forbidLinks),-1)<0)
83     {
84         addError(CRITICAL,"Unable to create label switched path in %s at line %d",
85             __FILE__,__LINE__);
86         return -1;
87     }
88
89     if (longListInit(&(lsp->path),-1)<0)
90     {
91         longListEnd(&(lsp->forbidLinks));
92         addError(CRITICAL,"Unable to create label switched path in %s at line %d",
93             __FILE__,__LINE__);
94         return -1;
95     }
96
97     if (longListInit(&(lsp->primPath),-1)<0)
98     {
99         longListEnd(&(lsp->path));
100        longListEnd(&(lsp->forbidLinks));
101        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
102            __FILE__,__LINE__);
103        return -1;
104    }
105
106    if (longListInit(&(lsp->backLSPIDs),-1)<0)
107    {
108        longListEnd(&(lsp->primPath));
109        longListEnd(&(lsp->path));
110        longListEnd(&(lsp->forbidLinks));
111        addError(CRITICAL,"Unable to create label switched path in %s at line %d",
112            __FILE__,__LINE__);
113        return -1;
114    }
115
116    memset(lsp->bw, 0, NB_OA * sizeof(double));
117    lsp->noContentionId=-1;    //very important
118
119    return 0;
120 }

```

4.12.3.30 int DBLspListDestroy (DBLSPList * list)

Definition at line 251 of file database.c.

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

```

252 {
253     if (list == NULL || list->cont == NULL)
254     {
255         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
256             __FILE__,__LINE__);
257         return -1;
258     }
259
260     free(list->cont);
261     free(list);
262
263     return 0;
264 }

```

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().

```

267 {
268     if (list == NULL || list->cont == NULL)
269     {
270         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
271                 __FILE__,__LINE__);
272         return -1;
273     }
274
275     free(list->cont);
276     list->cont = NULL;
277     list->size = 0;
278     list->top = 0;
279
280     return 0;
281 }
```

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().

```

224 {
225     void* ptr=NULL;
226
227     if (list == NULL)
228     {
229         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
230                 __FILE__,__LINE__);
231         return -1;
232     }
233
234     if (size == -1)
235         size = LSPLIST_INITSIZE;
236
237     if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
238     {
239         addError(CRITICAL,"Critical lack of memory in %s at line %d",
240                 __FILE__,__LINE__);
241         return -1;
242     }
243
244     list->size = size;
245     list->top = 0;
246     list->cont = ptr;
247
248     return 0;
249 }
```

4.12.3.33 int DBlspListInsert (DBLSPLList * list, DBLabelSwitchedPath * lsp)

Definition at line 283 of file database.c.

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

Referenced by DBaddLSP().

```

284 {
285     int a,b;
286     void *ptr=NULL;
287
288     if (list == NULL || list->cont == NULL || lsp == NULL)
289     {
290         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
291             __FILE__,__LINE__);
292         return -1;
293     }
294
295     // check the capacity of the list
296     if (list->top >= list->size)
297     {
298         if ((ptr = realloc(list->cont, list->size
299             * 2 * sizeof(DBLabelSwitchedPath*))) == NULL)
300         {
301             addError(CRITICAL,"Critical lack of memory in %s at line %d",
302                 __FILE__,__LINE__);
303             return -1;
304         }
305         else
306         {
307             list->cont=ptr;
308             list->size*=2;
309         }
310     }
311
312     // find the position in the list (to keep it sorted)
313     a = 0;
314     b = list->top-1;
315
316     // empty list or after the last elem
317     if (list->top == 0 || DBlspCompare(list->cont[b], lsp) >= 0)
318     {
319         list->cont[list->top++] = lsp;
320         return (list->top-1);
321     }
322
323     // before the first elem
324     if (DBlspCompare(lsp, list->cont[a]) >= 0)
325     {
326         memmove(list->cont+1, list->cont, (list->top)*sizeof(void*));
327         list->cont[0] = lsp;
328         list->top++;
329         return 0;
330     }
331
332     // now the insert position is inside [a,b[
333     while (b - a > 1)
334     {
335         int mid = (a + b)/2;
336         int ret = DBlspCompare(lsp, list->cont[mid]);
337
338         if (ret == 1)
339             b = mid;
340         else if (ret == -1)
341             a = mid;

```

```

342         else // if (ret == 0)
343         {
344             a = mid;
345             b = mid;
346         }
347     }
348
349     // now insert before b
350     memmove(list->cont+b+1, list->cont+b, (list->top - b)*sizeof(void*));
351     list->cont[b] = lsp;
352     list->top++;
353
354     return b;
355 }

```

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`.

```

194 {
195     DBLSPList *list=NULL;
196     void* ptr=NULL;
197
198     if ((list = calloc(1,sizeof(DBLSPList))) == NULL)
199     {
200         addError(CRITICAL,"Critical lack of memory in %s at line %d",
201             __FILE__,__LINE__);
202         return NULL;
203     }
204
205     if (size == -1)
206         size = LSPLIST_INITSIZE;
207
208     if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
209     {
210         addError(CRITICAL,"Critical lack of memory in %s at line %d",
211             __FILE__,__LINE__);
212         free(list);
213         return NULL;
214     }
215
216     list->size = size;
217     list->top = 0;
218     list->cont = ptr;
219
220     return list;
221 }

```

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`.

```

379 {
380     int a,b,index;

```

```

381
382     if (list == NULL || list->cont == NULL || lsp == NULL)
383     {
384         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
385             __FILE__,__LINE__);
386         return -1;
387     }
388
389     // find the position in the list
390     a = 0;
391     b = list->top-1;
392
393     // empty list
394     if (list->top == 0)
395     {
396         addError(WARNING,"Removing inexistent LSP in %s at line %d",
397             __FILE__,__LINE__);
398         return -1;
399     }
400
401     while (b - a > 1)
402     {
403         int mid = (a + b)/2;
404         int ret = DBLspCompare(lsp, list->cont[mid]);
405
406         if (ret == 1)
407             b = mid;
408         else if (ret == -1)
409             a = mid;
410         else // if (ret == 0)
411         {
412             a = mid;
413             b = mid;
414         }
415     }
416
417     if (DBLspCompare(lsp, list->cont[a]) == 0)
418     {
419         index = a;
420     }
421     else if (DBLspCompare(lsp, list->cont[b]) == 0)
422     {
423         index = b;
424     }
425     else // not found
426     {
427         addError(WARNING,"Removing inexistent LSP in %s at line %d",
428             __FILE__,__LINE__);
429         return -1;
430     }
431
432     // now delete index
433     memmove(list->cont + index, list->cont + index + 1, (list->top - index - 1)*sizeof(void*));
434     list->top--;
435
436     return 0;
437 }

```

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.

```

20 {
21     DBLabelSwitchedPath* lsp;
22
23     if ((lsp=calloc(1,sizeof(DBLabelSwitchedPath)))==NULL)
24     {
25         addError(CRITICAL,"Critical lack of memory in %s at line %d",
26                 __FILE__,__LINE__);
27         return NULL;
28     }
29
30     if (longListInit(&(lsp->forbidLinks),-1)<0)
31     {
32         free(lsp);
33         addError(CRITICAL,"Unable to create label switched path in %s at line %d",
34                 __FILE__,__LINE__);
35         return NULL;
36     }
37
38     if (longListInit(&(lsp->path),-1)<0)
39     {
40         longListEnd(&(lsp->forbidLinks));
41         free(lsp);
42         addError(CRITICAL,"Unable to create label switched path in %s at line %d",
43                 __FILE__,__LINE__);
44         return NULL;
45     }
46
47     if (longListInit(&(lsp->primPath),-1)<0)
48     {
49         longListEnd(&(lsp->path));
50         longListEnd(&(lsp->forbidLinks));
51         free(lsp);
52         addError(CRITICAL,"Unable to create label switched path in %s at line %d",
53                 __FILE__,__LINE__);
54         return NULL;
55     }
56
57     if (longListInit(&(lsp->backLSPIDs),-1)<0)
58     {
59         longListEnd(&(lsp->primPath));
60         longListEnd(&(lsp->path));
61         longListEnd(&(lsp->forbidLinks));
62         free(lsp);
63         addError(CRITICAL,"Unable to create label switched path in %s at line %d",
64                 __FILE__,__LINE__);
65         return NULL;
66     }
67
68     lsp->noContentionId=-1; //very important
69
70     return lsp;
71 }

```

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.

```

1338 {
1339     DataBase *dataBase=NULL;

```

```

1340
1341     if ((dataBase=calloc(1,sizeof(DataBase)))==NULL)
1342     {
1343         addError(CRITICAL,"Critical lack of memory in %s at line %d",
1344             __FILE__,__LINE__);
1345         return NULL;
1346     }
1347
1348     dataBase->id=ID;
1349
1350     if (DBnodeVecInit(&(dataBase->nodeVec),-1)<0)
1351     {
1352         addError(CRITICAL,"Unable to initialize the general node container in %s at line %d",
1353             __FILE__,__LINE__);
1354         free(dataBase);
1355         return NULL;
1356     }
1357
1358     if (DBlspVecInit(&(dataBase->lspVec),-1)<0)
1359     {
1360         addError(CRITICAL,"Unable to initialize the general LSP container in %s at line %d",
1361             __FILE__,__LINE__);
1362         DBnodeVecEnd(&(dataBase->nodeVec));
1363         free(dataBase);
1364         return NULL;
1365     }
1366
1367     if (DBlinkTabInit(&(dataBase->linkTab),-1)<0)
1368     {
1369         addError(CRITICAL,"Unable to initialize the general link container in %s at line %d",
1370             __FILE__,__LINE__);
1371         DBnodeVecEnd(&(dataBase->nodeVec));
1372         DBlspVecEnd(&(dataBase->lspVec));
1373         free(dataBase);
1374         return NULL;
1375     }
1376
1377     if (longVecInit(&(dataBase->linkSrcVec),LINKTAB_INITSIZE)<0)
1378     {
1379         addError(CRITICAL,"Unable to initialize the link id-src translator in %s at line %d",
1380             __FILE__,__LINE__);
1381         DBnodeVecEnd(&(dataBase->nodeVec));
1382         DBlspVecEnd(&(dataBase->lspVec));
1383         DBlinkTabEnd(&(dataBase->linkTab));
1384         free(dataBase);
1385         return NULL;
1386     }
1387
1388     if (longVecInit(&(dataBase->linkDstVec),LINKTAB_INITSIZE)<0)
1389     {
1390         addError(CRITICAL,"Unable to initialize the link id-dst translator in %s at line %d",
1391             __FILE__,__LINE__);
1392         DBnodeVecEnd(&(dataBase->nodeVec));
1393         DBlspVecEnd(&(dataBase->lspVec));
1394         DBlinkTabEnd(&(dataBase->linkTab));
1395         longVecEnd(&(dataBase->linkSrcVec));
1396         free(dataBase);
1397         return NULL;
1398     }
1399
1400     dataBase->nbNodes=0;
1401     dataBase->nbLinks=0;
1402
1403     return dataBase;
1404 }

```


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.

```

2314 {
2315     long i,j;
2316
2317     printf("Printing info about nodes ...\n");
2318     printf("-----\n");
2319
2320     for (i=0; i<db->nodeVec.size; i++)
2321     {
2322         if (db->nodeVec.cont[i])
2323         {
2324             printf("Node id : %ld\n", i);
2325             printf("-----\n");
2326             DBprintNode(db->nodeVec.cont[i]);
2327         }
2328     }
2329
2330     printf("\nPrinting info about links ...\n");
2331     printf("-----\n");
2332
2333     for (i=0; i<db->linkTab.size; i++)
2334         for (j=0; j<db->linkTab.size; j++)
2335         {
2336             if (db->linkTab.cont[i][j])
2337             {
2338                 printf("Link %ld-%ld (id = %ld)\n", i, j, DBgetLinkID(db, i, j));
2339                 printf("-----\n");
2340
2341                 DBprintLink(db->linkTab.cont[i][j]);
2342             }
2343         }
2344     }
2345 }
```

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().

```

1693 {
1694     int id,ret=0;
1695
1696     if (dataBase == NULL)
1697     {
1698         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1699                 __FILE__, __LINE__);
1700         return -1;
1701     }
1702
1703     if ((DBnodeVecGet(&(dataBase->nodeVec),src)==NULL) ||
```

```

1704         (DBnodeVecGet(&(dataBase->nodeVec),dst)==NULL) ||
1705         (DBlinkTabGet(&(dataBase->linkTab),src,dst)==NULL))
1706     {
1707         addError(CRITICAL,"Link doesn't exist or database unconsistency in %s at line %d",
1708             __FILE__,__LINE__);
1709         return -1;
1710     }
1711
1712     ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
1713     ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[dst]->inNeighb),src));
1714
1715     ANDERROR(ret,DBlinkTabRemove(&(dataBase->linkTab),src,dst));
1716
1717     id=DBgetLinkID(dataBase,src,dst);
1718     ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,0));
1719     ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,0));
1720
1721     while (dataBase->linkSrcVec.cont[dataBase->linkSrcVec.top-1] == 0)
1722         dataBase->linkSrcVec.top--;
1723
1724     if (ret<0)
1725     {
1726         addError(CRITICAL,"Link removal uncomplete in %s at line %d",
1727             __FILE__,__LINE__);
1728     }
1729
1730     dataBase->nbLinks--;
1731
1732     return ret;
1733 }

```

4.12.3.40 int DBremoveLSP (DataBase * dataBase, long id)

Definition at line 2065 of file database.c.

References addError(), ANDERROR, and DBlinkTabGet.

```

2066 {
2067     DBLabelSwitchedPath *lsp=NULL, *contentLSP=NULL;
2068     int i,ret=0;
2069     DBLink *lnk=NULL;
2070     LongVec isProcessed;
2071
2072     if (dataBase == NULL)
2073     {
2074         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2075             __FILE__,__LINE__);
2076         return -1;
2077     }
2078
2079     if ((lsp = DBlspVecGet(&(dataBase->lspVec), id)) == NULL)
2080     {
2081         addError(CRITICAL,"Trying to remove inexistent LSP (id = %ld) in %s at line %d",
2082             id,__FILE__,__LINE__);
2083         return -1;
2084     }
2085
2086     if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
2087     {
2088         addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
2089             __FILE__,__LINE__);
2090         return -1;
2091     }
2092

```

```

2093 #if defined SIMULATOR
2094     // Remove the LSP from each link list and update all the linkstates
2095     for (i=0;i<lsp->path.top-1;i++)
2096     {
2097         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
2098                         lsp->path.cont[i+1]);
2099         ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
2100         ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->path.cont[i],
2101                                         lsp->path.cont[i+1], &(lnk->state), lsp));
2102         isProcessed.cont[lnk->id] = 1;
2103     }
2104     if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
2105     {
2106         for (i=0;i<lsp->primPath.top-1;i++)
2107         {
2108             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
2109                             lsp->primPath.cont[i+1]);
2110             if (isProcessed.cont[lnk->id] == 0)
2111             {
2112                 ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->primPath.cont[i],
2113                                                 lsp->primPath.cont[i+1], &(lnk->state), lsp));
2114                 isProcessed.cont[lnk->id] = 1;
2115             }
2116         }
2117     }
2118 #elif defined AGENT
2119     // Remove the LSP to the link attached to the agent and update the linkstate
2120     for (i=0;i<lsp->path.top-1;i++)
2121     {
2122         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
2123                         lsp->path.cont[i+1]);
2124         ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
2125
2126         if (lsp->path.cont[i] == dataBase->id)
2127         {
2128             ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->path.cont[i],
2129                                             lsp->path.cont[i+1], &(lnk->state), lsp));
2130             isProcessed.cont[lnk->id] = 1;
2131         }
2132     }
2133     if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
2134     {
2135         for (i=0;i<lsp->primPath.top-1;i++)
2136         {
2137             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
2138                             lsp->primPath.cont[i+1]);
2139
2140             if (lsp->primPath.cont[i] == dataBase->id)
2141             {
2142                 if (isProcessed.cont[lnk->id] == 0)
2143                 {
2144                     ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->primPath.cont[i],
2145                                                     lsp->primPath.cont[i+1], &(lnk->state), lsp));
2146                 }
2147                 break;
2148             }
2149         }
2150     }
2151 #else
2152     // Generate an error;
2153     COMPILER_ERROR;
2154 #endif
2155     longVecEnd(&(isProcessed));
2156
2157     // remove the lsp from the global list
2158     ANDERROR(ret,DBlspVecRemove(&(dataBase->lspVec), id));

```

```

2160
2161     if (lsp->noContentionId>=0)
2162     {
2163         if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),lsp->noContentionId))==NULL)
2164         {
2165             addError(WARNING,"Unable to get no contention LSP in %s at line %d",
2166                     __FILE__,__LINE__);
2167             // not critical enough to abort
2168         }
2169         contentLSP->noContentionId=-1;
2170     }
2171
2172     // free the lsp
2173     DBlspDestroy(lsp);
2174
2175     if (ret<0)
2176     {
2177         addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
2178                 __FILE__,__LINE__);
2179     }
2180
2181     return ret;
2182 }

```

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.

```

1557 {
1558     DBNode *node=NULL;
1559     int ret=0;
1560
1561     if (dataBase == NULL)
1562     {
1563         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1564                 __FILE__,__LINE__);
1565         return -1;
1566     }
1567
1568     if ((node=DBnodeVecGet(&(dataBase->nodeVec),id)) == NULL)
1569     {
1570         addError(CRITICAL,"Trying to remove an inexistent node in %s at line %d",
1571                 __FILE__,__LINE__);
1572         return -1;
1573     }
1574
1575     // remember that DBremoveLink will update the neighbour list
1576     while(node->inNeighb.top > 0)
1577     {
1578         ANDERROR(ret,DBremoveLink(dataBase,node->inNeighb.cont[node->inNeighb.top-1],id));
1579     }
1580
1581     // remember that DBremoveLink will update the neighbour list
1582     while(node->outNeighb.top > 0)
1583     {
1584         ANDERROR(ret,DBremoveLink(dataBase,id,node->outNeighb.cont[node->outNeighb.top-1]));
1585     }
1586
1587     ANDERROR(ret,DBnodeVecRemove(&(dataBase->nodeVec),id));
1588 }

```

```

1589     if (ret<0)
1590     {
1591         addError(CRITICAL,"Node removal uncomplete in %s at line %d",
1592                 __FILE__,__LINE__);
1593     }
1594
1595     dataBase->nbLinks--;
1596
1597     return ret;
1598 }

```

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`.

```

2241 {
2242     DBLink *lnk=NULL;
2243
2244     if (dataBase == NULL || newLS == NULL)
2245     {
2246         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2247                 __FILE__,__LINE__);
2248         return -1;
2249     }
2250
2251     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2252     {
2253         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2254                 src,dst,__FILE__,__LINE__);
2255         return -1;
2256     }
2257
2258     if (DBlinkStateCopy(&(lnk->state), newLS)<0)
2259     {
2260         addError(CRITICAL,"Impossible to set linkstate on link (src = %ld, dst = %ld) in %s at line %d",
2261                 src,dst,__FILE__,__LINE__);
2262         return -1;
2263     }
2264
2265     return 0;
2266 }

```

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()`.

```

1327 {
1328     return updateLS(dataBase, src, dst, ls, lsp, REMOVE);
1329 }

```

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().

```
1322 {
1323     return updateLS(dataBase, src, dst, ls, lsp, SETUP);
1324 }
```

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().

```
839 {
840
841
842
843
844     DBLabelSwitchedPath* lsp, *primLSP;
845     int ret;
846
847     // check the arguments
848     if ((dataBase==NULL) || (newLS==NULL) || (oldLS==NULL) || (req==NULL))
849     {
850         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
851             __FILE__, __LINE__);
852         return -1;
853     }
854
855     // duplicate the LS
856     if (newLS != oldLS && DBlinkStateCopy(newLS, oldLS) < 0)
857     {
858         addError(CRITICAL,"Impossible to duplicate the linkState in %s at line %d",
859             __FILE__, __LINE__);
860         return -1;
861     }
862
863     // now build a false LSP satisfying the request ....
864     lsp = DBlspNew();
865     lsp->id = req->id;
866     lsp->precedence = req->precedence;
867     memcpy(lsp->bw, req->bw, NB_OA * sizeof(double));
868     longListCopy(&(lsp->forbidLinks), &(req->forbidLinks));
869
870     if (req->rerouteInfo.id >= 0)
871     {
872         lsp->noContentionId = req->rerouteInfo.id;
```

```

873     }
874
875     switch(req->type)
876     {
877         case PRIM:
878             lsp->type = PRIM;
879             lsp->primID = -1;
880             break;
881
882         case GLOBAL_BACK:
883         case LOCAL_BACK:
884             lsp->type = req->type;
885             lsp->primID = req->primID;
886
887             // look up the primary path ....
888             if ((primLSP = DBgetLSP(dataBase, lsp->primID)) == NULL)
889             {
890                 addError(CRITICAL, "Impossible to determine the primary path in %s at line %d",
891                     __FILE__, __LINE__);
892                 DBlspDestroy(lsp);
893                 return -1;
894             }
895
896             longListCopy(&(lsp->primPath), &(primLSP->path));
897
898             break;
899
900         default:
901             addError(CRITICAL, "Unknown request type (NULL) in %s at line %d",
902                 __FILE__, __LINE__);
903             DBlspDestroy(lsp);
904             return -1;
905     }
906
907     if (req->path.top < 2)
908     {
909         addError(CRITICAL, "Wrong path in request in %s at line %d",
910             __FILE__, __LINE__);
911         DBlspDestroy(lsp);
912         return -1;
913     }
914
915     if (longListCopy(&(lsp->path), &(req->path)) < 0)
916     {
917         addError(CRITICAL, "Impossible to duplicate path in %s at line %d",
918             __FILE__, __LINE__);
919         DBlspDestroy(lsp);
920         return -1;
921     }
922
923     ret = updateLS(dataBase, src, dst, newLS, lsp, op);
924
925     // clean up ....
926     DBlspDestroy(lsp);
927
928     return ret;
929 }

```

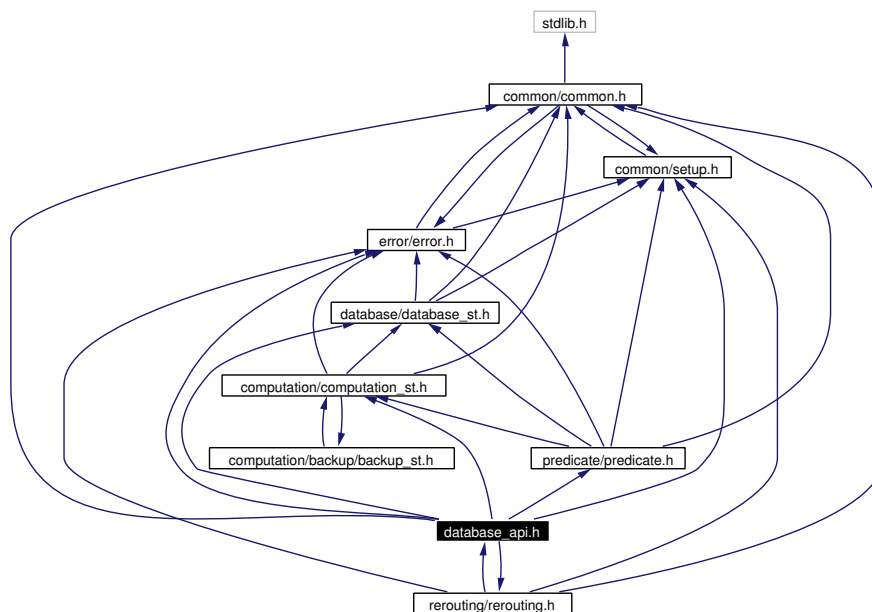
4.12.3.46 int updateLS (DataBase *, long, long, DBLinkState *, DBLabelSwitchedPath *, operation)

Referenced by DBupdateLSOnRemove(), DBupdateLSOnSetup(), and evalLS().

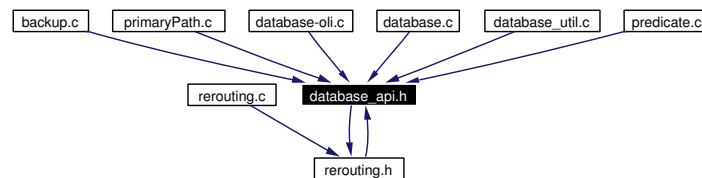
4.13 database_api.h File Reference

```
#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 DBevallLSPOnSetup (DataBase *, long, long, DBLinkState *, DBLinkState *, LSPRequest *)
- int DBevallLSPOnRemove (DataBase *, long, long, DBLinkState *, DBLinkState *, LSPRequest *)
- int DBupdateLSPOnSetup (DataBase *, long, long, DBLinkState *, DBLabelSwitchedPath *)
- int DBupdateLSPOnRemove (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.

```

1556 {
1557     DBLink* link=NULL;
1558     int ret=0;
1559
1560     if (dataBase == NULL || initLinkState==NULL
1561         || id <0 || src<0 || dst<0)
1562     {
1563         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1564             __FILE__,__LINE__);
1565         return -1;
1566     }
1567
1568     if (((id<dataBase->linkSrcVec.size) && (dataBase->linkSrcVec.cont[id]>0))
1569         ||
1570         ((id<dataBase->linkDstVec.size) && (dataBase->linkDstVec.cont[id]>0)))
1571     {
1572         addError(CRITICAL,"Trying to add a link with a reserved ID (ID=%ld) in %s at line %d",
1573             id,__FILE__,__LINE__);
1574         return -1;
1575     }
1576
1577     if ((link = DBlinkNew()) == NULL)
1578     {
1579         addError(CRITICAL,"Unable to create link in %s at line %d",
1580             __FILE__,__LINE__);
1581         return -1;
1582     }
1583
1584     link->id=id;
1585
1586     if (DBlinkStateCopy(&(link->state), initLinkState))
1587     {
1588         addError(CRITICAL,"Unable to create link in %s at line %d",
1589             __FILE__,__LINE__);
1590         DBlinkDestroy(link);
1591         return -1;
1592     }
1593
1594     if ((DBnodeVecGet(&(dataBase->nodeVec),src) == NULL) ||
1595         (DBnodeVecGet(&(dataBase->nodeVec),dst) == NULL))
1596     {
1597         addError(CRITICAL,"Source or destination node doesn't exist in %s at line %d",
1598             __FILE__,__LINE__);
1599         DBlinkDestroy(link);
1600         return -1;
1601     }
1602
1603     if (DBlinkTabSet(&(dataBase->linkTab),link,src,dst)<0)
1604     {
1605         addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %d",
1606             __FILE__,__LINE__);
1607         DBlinkDestroy(link);
1608         return -1;
1609     }
1610
1611     ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
1612     ANDERROR(ret,longListPushBack(&(dataBase->nodeVec.cont[dst]->inNeighb),src));
1613
1614     ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[src]->outNeighb)));
1615     ANDERROR(ret,longListSort(&(dataBase->nodeVec.cont[dst]->inNeighb)));
1616
1617     ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,src+1));
1618     ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,dst+1));
1619
1620     // Maximum non-null element
1621     dataBase->linkSrcVec.top = max(dataBase->linkSrcVec.top, id+1);
1622     dataBase->linkDstVec.top = dataBase->linkSrcVec.top;

```

```

1623
1624     if (ret<0)
1625     {
1626         addError(CRITICAL,"Link addition uncomplete in %s at line %d",
1627                 __FILE__,__LINE__);
1628     }
1629
1630     dataBase->nbLinks++;
1631
1632     return ret;
1633 }

```

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.

```

1680 {
1681     DBLabelSwitchedPath *newLSP, *contentLSP=NULL;
1682     DBLSPList *lspList;
1683     int i,ret=0;
1684     DBLink *lnk=NULL;
1685     LongVec isProcessed;
1686     double rerouteGain[NB_OA];
1687     bool allowLSP=TRUE;
1688     #if defined SIMULATOR
1689         LongList idList;
1690     #elif defined AGENT
1691         int j;
1692         bool inPath=FALSE;
1693     #endif
1694
1695     #if defined LINUX && defined TIME2
1696         struct timezone tz;
1697         struct timeval t1,t2;
1698     #endif
1699
1700     if (dataBase == NULL || lsp==NULL)
1701     {
1702         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1703                 __FILE__,__LINE__);
1704         return -1;
1705     }
1706
1707     #if defined LINUX && defined TIME2
1708         gettimeofday(&t1, &tz);
1709     #endif
1710
1711     if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
1712     {
1713         addError(CRITICAL,"Unable to initialize vector of longs in %s at line %d",
1714                 __FILE__,__LINE__);
1715         return -1;
1716     }
1717

```

```

1718     memset(rerouteGain,0,NB_OA*sizeof(double));
1719
1720     // Check if establishment is possible
1721 #if defined SIMULATOR
1722     if (longListInit(&(idList),-1)<0)
1723     {
1724         addError(CRITICAL,"Unable to initialize list of longs in %s at line %d",
1725             __FILE__,__LINE__);
1726         return -1;
1727     }
1728     for (i=0;(i<lsp->path.top-1) && allowLSP;i++)
1729     {
1730         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
1731             lsp->path.cont[i+1]);
1732         allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->path.cont[i],lsp->path.cont[i+1],
1733             &(lnk->state),lsp,rerouteGain);
1734         if ((lspList=DBgetLinkLSPs(dataBase,lsp->path.cont[i],lsp->path.cont[i+1]))==NULL)
1735         {
1736             addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1737                 __FILE__,__LINE__);
1738             longListEnd(&(idList));
1739             longVecEnd(&(isProcessed));
1740             return -1;
1741         }
1742         idList.top=0;
1743         if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
1744         {
1745             addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1746                 __FILE__,__LINE__);
1747             longListEnd(&(idList));
1748             longVecEnd(&(isProcessed));
1749             return -1;
1750         }
1751         if (longListMerge(&(idList),preemptList,preemptList)<0)
1752         {
1753             addError(CRITICAL,"Unable to merge lists of longs in %s at line %d",
1754                 __FILE__,__LINE__);
1755             longListEnd(&(idList));
1756             longVecEnd(&(isProcessed));
1757             return -1;
1758         }
1759         isProcessed.cont[lnk->id] = 1;
1760     }
1761     if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
1762     {
1763         for (i=0;(i<lsp->primPath.top-1) && allowLSP;i++)
1764         {
1765             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
1766                 lsp->primPath.cont[i+1]);
1767             if (isProcessed.cont[lnk->id] == 0)
1768             {
1769                 allowLSP= allowLSP && isValidLSPLink(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1],
1770                     &(lnk->state),lsp,rerouteGain);
1771                 if ((lspList=DBgetLinkLSPs(dataBase,lsp->primPath.cont[i],lsp->primPath.cont[i+1]))==NULL)
1772                 {
1773                     addError(CRITICAL,"Unable to get the list of LSPs carried by the link in %s at line %d",
1774                         __FILE__,__LINE__);
1775                     longListEnd(&(idList));
1776                     longVecEnd(&(isProcessed));
1777                     return -1;
1778                 }
1779                 idList.top=0;
1780                 if (chooseReroutedLSPs(lsp->precedence,&(lnk->state),lspList,rerouteGain,&(idList))<0)
1781                 {
1782                     addError(CRITICAL,"Unable choose LSPs for rerouting in %s at line %d",
1783                         __FILE__,__LINE__);
1784                     longListEnd(&(idList));

```

```

1785         longVecEnd(&(isProcessed));
1786         return -1;
1787     }
1788     if (longListMerge(&(idList), preemptList, preemptList) < 0)
1789     {
1790         addError(CRITICAL, "Unable to merge lists of longs in %s at line %d",
1791                 __FILE__, __LINE__);
1792         longListEnd(&(idList));
1793         longVecEnd(&(isProcessed));
1794         return -1;
1795     }
1796     isProcessed.cont[lnk->id] = 1;
1797 }
1798 }
1799 }
1800 longListEnd(&(idList));
1801 #elif defined AGENT
1802 for (i=0; i<lsp->path.top-1) && (lsp->path.cont[i]!=dataBase->id); i++);
1803
1804 if (i<lsp->path.top-1)
1805 {
1806     lnk=DBlinkTabGet(&(dataBase->linkTab), lsp->path.cont[i],
1807                    lsp->path.cont[i+1]);
1808     allowLSP= allowLSP && isValidLSPLink(dataBase, lsp->path.cont[i], lsp->path.cont[i+1],
1809                                         &(lnk->state), lsp, rerouteGain);
1810     if ((lspList=DBgetLinkLSPs(dataBase, lsp->path.cont[i], lsp->path.cont[i+1]))==NULL)
1811     {
1812         addError(CRITICAL, "Unable to get the list of LSPs carried by the link in %s at line %d",
1813                 __FILE__, __LINE__);
1814         longVecEnd(&(isProcessed));
1815     }
1816     if (chooseReroutedLSPs(lsp->precedence, &(lnk->state), lspList, rerouteGain, preemptList) < 0)
1817     {
1818         addError(CRITICAL, "Unable choose LSPs for rerouting in %s at line %d",
1819                 __FILE__, __LINE__);
1820         longVecEnd(&(isProcessed));
1821         return -1;
1822     }
1823     isProcessed.cont[lnk->id] = 1;
1824     inPath=TRUE;
1825 }
1826 if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
1827 {
1828     for (j=0; j<lsp->primPath.top-1) && (lsp->primPath.cont[j]!=dataBase->id); j++);
1829
1830     if (j<lsp->primPath.top-1)
1831     {
1832         lnk=DBlinkTabGet(&(dataBase->linkTab), lsp->primPath.cont[j],
1833                        lsp->primPath.cont[j+1]);
1834         if (isProcessed.cont[lnk->id] == 0)
1835         {
1836             allowLSP= allowLSP && isValidLSPLink(dataBase, lsp->primPath.cont[j], lsp->primPath.cont[j+1],
1837                                                 &(lnk->state), lsp, rerouteGain);
1838             if ((lspList=DBgetLinkLSPs(dataBase, lsp->primPath.cont[j], lsp->primPath.cont[j+1]))==NULL)
1839             {
1840                 addError(CRITICAL, "Unable to get the list of LSPs carried by the link in %s at line %d",
1841                         __FILE__, __LINE__);
1842                 longVecEnd(&(isProcessed));
1843             }
1844             if (chooseReroutedLSPs(lsp->precedence, &(lnk->state), lspList, rerouteGain, preemptList) < 0)
1845             {
1846                 addError(CRITICAL, "Unable choose LSPs for rerouting in %s at line %d",
1847                         __FILE__, __LINE__);
1848                 longVecEnd(&(isProcessed));
1849                 return -1;
1850             }
1851             isProcessed.cont[lnk->id] = 1;

```

```

1852         }
1853         inPath=TRUE;
1854     }
1855 }
1856 if (!inPath)
1857 {
1858     addError(CRITICAL,"Agent not concerned by this LSP in %s at line %d",
1859         __FILE__,__LINE__);
1860     longVecEnd(&(isProcessed));
1861     return -1;
1862 }
1863 #else
1864     // Generate an error;
1865     COMPILE_ERROR;
1866 #endif
1867
1868 if (!allowLSP)
1869 {
1870     addError(CRITICAL,"LSP refused by the predicate in %s at line %d",
1871         __FILE__,__LINE__);
1872     longVecEnd(&(isProcessed));
1873     return -1;
1874 }
1875
1876
1877 if ((newLSP=DBlspNew())==NULL)
1878 {
1879     addError(CRITICAL,"Unable to create LSP in %s at line %d",
1880         __FILE__,__LINE__);
1881     longVecEnd(&(isProcessed));
1882     return -1;
1883 }
1884
1885 if (DBlspCopy(newLSP,lsp)<0)
1886 {
1887     addError(CRITICAL,"Unable to create a valid LSP copy in %s at line %d",
1888         __FILE__,__LINE__);
1889     DBlspDestroy(newLSP);
1890     longVecEnd(&(isProcessed));
1891     return -1;
1892 }
1893
1894 if (DBlspVecSet(&(dataBase->lspVec),newLSP,newLSP->id)<0)
1895 {
1896     addError(CRITICAL,"Unable to insert LSP in the general LSP container in %s at line %d",
1897         __FILE__,__LINE__);
1898     DBlspDestroy(newLSP);
1899     longVecEnd(&(isProcessed));
1900     return -1;
1901 }
1902
1903 if (newLSP->noContentionId>=0)
1904 {
1905     if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),newLSP->noContentionId))==NULL)
1906     {
1907         addError(WARNING,"Unable to get no contention LSP in %s at line %d",
1908             __FILE__,__LINE__);
1909         newLSP->noContentionId=-1;
1910         // not critical enough to abort
1911     }
1912     else
1913     {
1914         contentLSP->noContentionId=newLSP->id;
1915     }
1916 }
1917
1918 for (i=0;i<isProcessed.size;i++)

```

```

1919     {
1920         isProcessed.cont[i]=0;
1921     }
1922
1923
1924 #if defined SIMULATOR
1925     // Add the LSP to each link list and update all the linkstates (only once !!!!!)
1926     for (i=0;i<newLSP->path.top-1;i++)
1927     {
1928         lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
1929             newLSP->path.cont[i+1]);
1930         ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
1931         ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->path.cont[i],
1932             newLSP->path.cont[i+1], &(lnk->state), newLSP));
1933         isProcessed.cont[lnk->id] = 1;
1934     }
1935     if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
1936     {
1937         for (i=0;i<newLSP->primPath.top-1;i++)
1938         {
1939             lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
1940                 newLSP->primPath.cont[i+1]);
1941             if (isProcessed.cont[lnk->id] == 0)
1942             {
1943                 ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->primPath.cont[i],
1944                     newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
1945                 isProcessed.cont[lnk->id] = 1;
1946             }
1947         }
1948     }
1949 #elif defined AGENT
1950     // Add the LSP to the link attached to the agent and update the linkstate
1951     for (i=0;i<newLSP->path.top-1;i++)
1952     {
1953         lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->path.cont[i],
1954             newLSP->path.cont[i+1]);
1955         ANDERROR(ret,DBlspListInsert(&(lnk->lspList),newLSP));
1956
1957         if (newLSP->path.cont[i] == dataBase->id)
1958         {
1959             ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->path.cont[i],
1960                 newLSP->path.cont[i+1], &(lnk->state), newLSP));
1961             isProcessed.cont[lnk->id] = 1;
1962         }
1963     }
1964     if ((newLSP->type == GLOBAL_BACK) || (newLSP->type == LOCAL_BACK))
1965     {
1966         for (i=0;i<newLSP->primPath.top-1;i++)
1967         {
1968             lnk=DBlinkTabGet(&(dataBase->linkTab),newLSP->primPath.cont[i],
1969                 newLSP->primPath.cont[i+1]);
1970
1971             if (newLSP->primPath.cont[i] == dataBase->id)
1972             {
1973                 if (isProcessed.cont[lnk->id] == 0)
1974                 {
1975                     ANDERROR(ret,DBupdateLSONSetup(dataBase, newLSP->primPath.cont[i],
1976                         newLSP->primPath.cont[i+1], &(lnk->state), newLSP));
1977                 }
1978                 break;
1979             }
1980         }
1981     }
1982 #else
1983     // Generate an error;
1984     COMPILER_ERROR;
1985 #endif

```

```

1986
1987     longVecEnd(&(isProcessed));
1988
1989 #if defined LINUX && defined TIME2
1990     gettimeofday(&t2, &tz);
1991     fprintf(stderr, "Time to add a new LSP : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
1992             (t2.tv_usec - t1.tv_usec) / 1000.0);
1993 #endif
1994
1995
1996     if (ret<0)
1997     {
1998         addError(CRITICAL,"LSP addition uncomplete in %s at line %d",
1999             __FILE__, __LINE__);
2000     }
2001
2002     return ret;
2003 }

```

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.

```

1467 {
1468     DBNode *node=NULL;
1469
1470     if (dataBase == NULL)
1471     {
1472         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1473             __FILE__, __LINE__);
1474         return -1;
1475     }
1476
1477     if ((node=DBnodeNew()) == NULL)
1478     {
1479         addError(CRITICAL,"Unable to create node in %s at line %d",
1480             __FILE__, __LINE__);
1481         return -1;
1482     }
1483
1484     node->id=id;
1485
1486     if (DBnodeVecSet(&(dataBase->nodeVec),node,id) < 0)
1487     {
1488         addError(CRITICAL,"Unable to insert a new node in the general node container in %s at line %d",
1489             __FILE__, __LINE__);
1490         DBnodeDestroy(node);
1491         return -1;
1492     }
1493
1494     dataBase->nbNodes++;
1495
1496     return 0;
1497 }

```

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.

```

1350 {
1351     if (dataBase == NULL)
1352     {
1353         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
1354             __FILE__, __LINE__);
1355         return -1;
1356     }
1357
1358     DBnodeVecEnd(&(dataBase->nodeVec));
1359     DBlspVecEnd(&(dataBase->lspVec));
1360     DBlinkTabEnd(&(dataBase->linkTab));
1361     longVecEnd(&(dataBase->linkSrcVec));
1362     longVecEnd(&(dataBase->linkDstVec));
1363
1364     free(dataBase);
1365
1366     return 0;
1367 }
```

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().

```

1260 {
1261     return evalLS(dataBase, src, dst, newLS, oldLS, req, REMOVE);
1262 }
```

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().

```

1254 {
1255     return evalLS(dataBase, src, dst, newLS, oldLS, req, SETUP);
1256 }
```

4.13.1.7 long DBgetID (DataBase *)

Definition at line 1369 of file database-oli.c.

References addError(), CRITICAL, and DataBase_::id.

```

1370 {
1371     if (dataBase == NULL)
1372     {
1373         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1374             __FILE__,__LINE__);
1375         return -1;
1376     }
1377
1378     return dataBase->id;
1379 }

```

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().

```

1422 {
1423     long ret;
1424
1425     if (dataBase == NULL)
1426     {
1427         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1428             __FILE__,__LINE__);
1429         return -1;
1430     }
1431
1432     if (longVecGet(&(dataBase->linkDstVec),id,&ret)<0)
1433     {
1434         addError(CRITICAL,"Inexistent link in %s at line %d",
1435             __FILE__,__LINE__);
1436         return -1;
1437     }
1438
1439     return (ret-1);
1440 }

```

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().

```

1382 {
1383     DBLink *lnk=NULL;
1384
1385     if (dataBase == NULL || src < 0 || dst < 0)
1386     {
1387         addError(CRITICAL,"Bad argument (NULL or negative value) in %s at line %d",
1388             __FILE__,__LINE__);
1389         return -1;
1390     }
1391
1392     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst))==NULL)
1393     {
1394         return -1;
1395     }

```

```
1396
1397     return lnk->id;
1398 }
```

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()`.

```
2139 {
2140     DBLink *lnk=NULL;
2141
2142     if (dataBase == NULL)
2143     {
2144         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2145                 __FILE__,__LINE__);
2146         return NULL;
2147     }
2148
2149     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2150     {
2151         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2152                 src,dst,__FILE__,__LINE__);
2153         return NULL;
2154     }
2155
2156     return &(lnk->lspList);
2157 }
```

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()`.

```
1401 {
1402     long ret;
1403
1404     if (dataBase == NULL)
1405     {
1406         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1407                 __FILE__,__LINE__);
1408         return -1;
1409     }
1410
1411     if (longVecGet(&(dataBase->linkSrcVec),id,&ret)<0)
1412     {
1413         addError(CRITICAL,"Inexistent link in %s at line %d",
1414                 __FILE__,__LINE__);
1415         return -1;
1416     }
1417
1418     return (ret-1);
1419 }
```

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().

```

2160 {
2161     DBLink *lnk=NULL;
2162
2163     if (dataBase == NULL)
2164     {
2165         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2166             __FILE__, __LINE__);
2167         return NULL;
2168     }
2169
2170     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2171     {
2172         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2173             src,dst,__FILE__, __LINE__);
2174         return NULL;
2175     }
2176
2177     return &(lnk->state);
2178 }

```

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().

```

2126 {
2127     if (dataBase == NULL)
2128     {
2129         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2130             __FILE__, __LINE__);
2131         return NULL;
2132     }
2133
2134     return DBlspVecGet(&(dataBase->lspVec), id);
2135 }

```

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().

```

1455 {
1456     if (dataBase == NULL)
1457     {
1458         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1459             __FILE__, __LINE__);

```

```
1460         return -1;
1461     }
1462
1463     return dataBase->nodeVec.top-1;
1464 }
```

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\(\)](#).

```
1544 {
1545     if (dataBase == NULL)
1546     {
1547         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
1548                 __FILE__, __LINE__);
1549         return -1;
1550     }
1551
1552     return dataBase->nbLinks;
1553 }
```

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\(\)](#).

```
1443 {
1444     if (dataBase == NULL)
1445     {
1446         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
1447                 __FILE__, __LINE__);
1448         return -1;
1449     }
1450
1451     return dataBase->nbNodes;
1452 }
```

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\(\)](#).

```
2210 {
2211     DBNode *node=NULL;
2212
2213     if (dataBase == NULL)
2214     {
2215         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
```

```

2216         __FILE__, __LINE__);
2217     return NULL;
2218 }
2219
2220 if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
2221 {
2222     addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
2223             id,__FILE__,__LINE__);
2224     return NULL;
2225 }
2226
2227 return (&(node->inNeighb));
2228 }

```

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().

```

2232 {
2233     DBNode *node=NULL;
2234
2235     if (dataBase == NULL)
2236     {
2237         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2238                 __FILE__,__LINE__);
2239         return NULL;
2240     }
2241
2242     if ((node=DBnodeVecGet(&(dataBase->nodeVec), id)) == NULL)
2243     {
2244         addError(CRITICAL,"Node don't exist (id = %ld) in %s at line %d",
2245                 id,__FILE__,__LINE__);
2246         return NULL;
2247     }
2248
2249     return (&(node->outNeighb));
2250 }

```

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().

```

661 {
662     int i,j,ret=0;
663
664     if (dst == NULL || src == NULL)
665     {
666         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
667                 __FILE__,__LINE__);
668         return -1;
669     }

```

```

670
671     dst->color=src->color;
672     memcpy(&(dst->cap),&(src->cap),NB_OA * sizeof(double));
673     memcpy(&(dst->rbw),&(src->rbw),NB_OA * NB_PREEMPTION * sizeof(double));
674     memcpy(&(dst->pbw),&(src->pbw),NB_OA * NB_PREEMPTION * sizeof(double));
675
676     for (i=0;(i<NB_OA && ret>=0);i++)
677         for (j=0;(j<NB_PREEMPTION && ret>=0);j++)
678             {
679                 ANDERROR(ret,dblVecCopy(&(dst->bbw[i][j]),&(src->bbw[i][j])));
680                 ANDERROR(ret,dblVecCopy(&(dst->remoteBbw[i][j]),&(src->remoteBbw[i][j])));
681                 ANDERROR(ret,dblVecCopy(&(dst->fbw[i][j]),&(src->fbw[i][j])));
682                 ANDERROR(ret,dblVecCopy(&(dst->remoteFbw[i][j]),&(src->remoteFbw[i][j])));
683             }
684
685     if (ret<0)
686     {
687         addError(CRITICAL,"Link state copy uncomplete in %s at line %d",
688                 __FILE__,__LINE__);
689     }
690
691     return ret;
692 }

```

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()`.

```

614 {
615     int i,j;
616
617     if (ls == NULL)
618     {
619         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
620                 __FILE__,__LINE__);
621         return -1;
622     }
623
624     for (i=0;i<NB_OA;i++)
625         for (j=0;j<NB_PREEMPTION;j++)
626             {
627                 dblVecEnd(&(ls->bbw[i][j]));
628                 dblVecEnd(&(ls->remoteBbw[i][j]));
629                 dblVecEnd(&(ls->fbw[i][j]));
630                 dblVecEnd(&(ls->remoteFbw[i][j]));
631             }
632     free(ls);
633
634     return 0;
635 }

```

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 isValidRequestLink().

```

638 {
639     int i,j;
640
641     if (ls == NULL)
642     {
643         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
644                 __FILE__,__LINE__);
645         return -1;
646     }
647
648     for (i=0;i<NB_OA;i++)
649         for (j=0;j<NB_PREEMPTION;j++)
650         {
651             dblVecEnd(&(ls->bbw[i][j]));
652             dblVecEnd(&(ls->remoteBbw[i][j]));
653             dblVecEnd(&(ls->fbw[i][j]));
654             dblVecEnd(&(ls->remoteFbw[i][j]));
655         }
656
657     return 0;
658 }

```

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().

```

531 {
532     int i,j,k,l;
533
534     if (ls == NULL)
535     {
536         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
537                 __FILE__,__LINE__);
538         return -1;
539     }
540
541     memset(ls, 0, sizeof(DBLinkState));
542
543     for (i=0;i<NB_OA;i++)
544         for (j=0;j<NB_PREEMPTION;j++)
545         {
546             if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
547             {
548                 for (k=i;k>=0;k++)
549                     for (l=j-1;l>=0;l++)
550                     {
551                         dblVecEnd(&(ls->bbw[k][l]));
552                         dblVecEnd(&(ls->remoteBbw[k][l]));
553                         dblVecEnd(&(ls->fbw[k][l]));
554                         dblVecEnd(&(ls->remoteFbw[k][l]));
555                     }
556                 addError(CRITICAL,"Unable to create link state in %s at line %d",
557                         __FILE__,__LINE__);
558                 return -1;
559             }
560         }
561     }
562 }

```



```

560         else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
561         {
562             dblVecEnd(&(ls->bbw[i][j]));
563             for (k=i;k>=0;k++)
564                 for (l=j-1;l>=0;l++)
565                 {
566                     dblVecEnd(&(ls->bbw[k][l]));
567                     dblVecEnd(&(ls->remoteBbw[k][l]));
568                     dblVecEnd(&(ls->fbw[k][l]));
569                     dblVecEnd(&(ls->remoteFbw[k][l]));
570                 }
571             addError(CRITICAL,"Unable to create link state in %s at line %d",
572                     __FILE__,__LINE__);
573             return -1;
574         }
575         else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
576         {
577             dblVecEnd(&(ls->bbw[i][j]));
578             dblVecEnd(&(ls->remoteBbw[i][j]));
579             for (k=i;k>=0;k++)
580                 for (l=j-1;l>=0;l++)
581                 {
582                     dblVecEnd(&(ls->bbw[k][l]));
583                     dblVecEnd(&(ls->remoteBbw[k][l]));
584                     dblVecEnd(&(ls->fbw[k][l]));
585                     dblVecEnd(&(ls->remoteFbw[k][l]));
586                 }
587             addError(CRITICAL,"Unable to create link state in %s at line %d",
588                     __FILE__,__LINE__);
589             return -1;
590         }
591         else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
592         {
593             dblVecEnd(&(ls->bbw[i][j]));
594             dblVecEnd(&(ls->remoteBbw[i][j]));
595             dblVecEnd(&(ls->fbw[i][j]));
596             for (k=i;k>=0;k++)
597                 for (l=j-1;l>=0;l++)
598                 {
599                     dblVecEnd(&(ls->bbw[k][l]));
600                     dblVecEnd(&(ls->remoteBbw[k][l]));
601                     dblVecEnd(&(ls->fbw[k][l]));
602                     dblVecEnd(&(ls->remoteFbw[k][l]));
603                 }
604             addError(CRITICAL,"Unable to create link state in %s at line %d",
605                     __FILE__,__LINE__);
606             return -1;
607         }
608     }
609     return 0;
610 }
611 }

```

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.

```

445 {
446     DBLinkState* ls;
447     int i,j,k,l;
448

```

```

449     if ((ls=calloc(1,sizeof(DBLinkState)))==NULL)
450     {
451         addError(CRITICAL,"Critical lack of memory in %s at line %d",
452             __FILE__,__LINE__);
453         return NULL;
454     }
455
456     for (i=0;i<NB_OA;i++)
457         for (j=0;j<NB_PREEMPTION;j++)
458         {
459             if (dblVecInit(&(ls->bbw[i][j]),-1)<0)
460             {
461                 for (k=i;k>=0;k--)
462                     for (l=j-1;l>=0;l--)
463                     {
464                         dblVecEnd(&(ls->bbw[k][l]));
465                         dblVecEnd(&(ls->remoteBbw[k][l]));
466                         dblVecEnd(&(ls->fbw[k][l]));
467                         dblVecEnd(&(ls->remoteFbw[k][l]));
468                     }
469                 free(ls);
470                 addError(CRITICAL,"Unable to create link state in %s at line %d",
471                     __FILE__,__LINE__);
472                 return NULL;
473             }
474             else if (dblVecInit(&(ls->remoteBbw[i][j]),-1)<0)
475             {
476                 dblVecEnd(&(ls->bbw[i][j]));
477                 for (k=i;k>=0;k--)
478                     for (l=j-1;l>=0;l--)
479                     {
480                         dblVecEnd(&(ls->bbw[k][l]));
481                         dblVecEnd(&(ls->remoteBbw[k][l]));
482                         dblVecEnd(&(ls->fbw[k][l]));
483                         dblVecEnd(&(ls->remoteFbw[k][l]));
484                     }
485                 free(ls);
486                 addError(CRITICAL,"Unable to create link state in %s at line %d",
487                     __FILE__,__LINE__);
488                 return NULL;
489             }
490             else if (dblVecInit(&(ls->fbw[i][j]),-1)<0)
491             {
492                 dblVecEnd(&(ls->bbw[i][j]));
493                 dblVecEnd(&(ls->remoteBbw[i][j]));
494                 for (k=i;k>=0;k--)
495                     for (l=j-1;l>=0;l--)
496                     {
497                         dblVecEnd(&(ls->bbw[k][l]));
498                         dblVecEnd(&(ls->remoteBbw[k][l]));
499                         dblVecEnd(&(ls->fbw[k][l]));
500                         dblVecEnd(&(ls->remoteFbw[k][l]));
501                     }
502                 free(ls);
503                 addError(CRITICAL,"Unable to create link state in %s at line %d",
504                     __FILE__,__LINE__);
505                 return NULL;
506             }
507             else if (dblVecInit(&(ls->remoteFbw[i][j]),-1)<0)
508             {
509                 dblVecEnd(&(ls->bbw[i][j]));
510                 dblVecEnd(&(ls->remoteBbw[i][j]));
511                 dblVecEnd(&(ls->fbw[i][j]));
512                 for (k=i;k>=0;k--)
513                     for (l=j-1;l>=0;l--)
514                     {
515                         dblVecEnd(&(ls->bbw[k][l]));

```

```

516             dblVecEnd(&(ls->remoteBbw[k][1]));
517             dblVecEnd(&(ls->fbw[k][1]));
518             dblVecEnd(&(ls->remoteFbw[k][1]));
519         }
520         free(ls);
521         addError(CRITICAL,"Unable to create link state in %s at line %d",
522             __FILE__,__LINE__);
523         return NULL;
524     }
525 }
526
527 return ls;
528 }

```

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().

```

358 {
359     if (LSPa->precedence > LSPb->precedence)
360         return 1;
361     else if (LSPa->precedence < LSPb->precedence)
362         return -1;
363     else if (LSPa->bw[0] > LSPb->bw[0])
364         return 1;
365     else if (LSPa->bw[0] < LSPb->bw[0])
366         return -1;
367     else
368     {
369         if (LSPa->id < LSPb->id)
370             return 1;
371         else if (LSPa->id > LSPb->id)
372             return -1;
373     }
374     return 0;
375 }
376 }

```

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().

```

158 {
159     int ret=0;
160
161     if (dst == NULL || src==NULL)
162     {

```

```

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     {

```

```

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 }

```

4.13.1.28 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`.

```

74 {
75     if (lsp == NULL)
76     {
77         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
78                 __FILE__, __LINE__);
79         return -1;
80     }
81
82     if (longListInit(&(lsp->forbidLinks), -1) < 0)
83     {
84         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
85                 __FILE__, __LINE__);
86         return -1;
87     }
88
89     if (longListInit(&(lsp->path), -1) < 0)
90     {
91         longListEnd(&(lsp->forbidLinks));
92         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
93                 __FILE__, __LINE__);
94         return -1;
95     }
96
97     if (longListInit(&(lsp->primPath), -1) < 0)
98     {
99         longListEnd(&(lsp->path));
100         longListEnd(&(lsp->forbidLinks));
101         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
102                 __FILE__, __LINE__);
103         return -1;
104     }
105
106     if (longListInit(&(lsp->backLSPIDs), -1) < 0)
107     {
108         longListEnd(&(lsp->primPath));
109         longListEnd(&(lsp->path));
110         longListEnd(&(lsp->forbidLinks));
111         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
112                 __FILE__, __LINE__);
113         return -1;
114     }
115
116     memset(lsp->bw, 0, NB_OA * sizeof(double));

```

```
117     lsp->noContentionId=-1;    //very important
118
119     return 0;
120 }
```

4.13.1.29 int DBLspListDestroy ([DBLSPList](#) *)

Definition at line 251 of file database-oli.c.

References [addError\(\)](#), [DBLSPList::cont](#), [CRITICAL](#), and [free](#).

```
252 {
253     if (list == NULL || list->cont == NULL)
254     {
255         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
256                 __FILE__,__LINE__);
257         return -1;
258     }
259
260     free(list->cont);
261     free(list);
262
263     return 0;
264 }
```

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\(\)](#).

```
267 {
268     if (list == NULL || list->cont == NULL)
269     {
270         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
271                 __FILE__,__LINE__);
272         return -1;
273     }
274
275     free(list->cont);
276     list->cont = NULL;
277     list->size = 0;
278     list->top = 0;
279
280     return 0;
281 }
```

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\(\)](#).

```

224 {
225     void* ptr=NULL;
226
227     if (list == NULL)
228     {
229         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
230                 __FILE__,__LINE__);
231         return -1;
232     }
233
234     if (size == -1)
235         size = LSPLIST_INITSIZE;
236
237     if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
238     {
239         addError(CRITICAL,"Critical lack of memory in %s at line %d",
240                 __FILE__,__LINE__);
241         return -1;
242     }
243
244     list->size = size;
245     list->top = 0;
246     list->cont = ptr;
247
248     return 0;
249 }

```

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().

```

284 {
285     int a,b;
286     void *ptr=NULL;
287
288     if (list == NULL || list->cont == NULL || lsp == NULL)
289     {
290         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
291                 __FILE__,__LINE__);
292         return -1;
293     }
294
295     // check the capacity of the list
296     if (list->top >= list->size)
297     {
298         if ((ptr = realloc(list->cont, list->size
299                             * 2 * sizeof(DBLabelSwitchedPath*))) == NULL)
300         {
301             addError(CRITICAL,"Critical lack of memory in %s at line %d",
302                     __FILE__,__LINE__);
303             return -1;
304         }
305         else
306         {
307             list->cont=ptr;
308             list->size*=2;
309         }
310     }
311 }

```

```

312 // find the position in the list (to keep it sorted)
313 a = 0;
314 b = list->top-1;
315
316 // empty list or after the last elem
317 if (list->top == 0 || DBlspCompare(list->cont[b], lsp) >= 0)
318 {
319     list->cont[list->top++] = lsp;
320     return (list->top-1);
321 }
322
323 // before the first elem
324 if (DBlspCompare(lsp, list->cont[a]) >= 0)
325 {
326     memmove(list->cont+1, list->cont, (list->top)*sizeof(void*));
327     list->cont[0] = lsp;
328     list->top++;
329     return 0;
330 }
331
332 // now the insert position is inside ]a,b[
333 while (b - a > 1)
334 {
335     int mid = (a + b)/2;
336     int ret = DBlspCompare(lsp, list->cont[mid]);
337
338     if (ret == 1)
339         b = mid;
340     else if (ret == -1)
341         a = mid;
342     else // if (ret == 0)
343     {
344         a = mid;
345         b = mid;
346     }
347 }
348
349 // now insert before b
350 memmove(list->cont+b+1, list->cont+b, (list->top - b)*sizeof(void*));
351 list->cont[b] = lsp;
352 list->top++;
353
354 return b;
355 }

```

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.

```

194 {
195     DBLSPList *list=NULL;
196     void* ptr=NULL;
197
198     if ((list = calloc(1,sizeof(DBLSPList))) == NULL)
199     {
200         addError(CRITICAL,"Critical lack of memory in %s at line %d",
201                 __FILE__,__LINE__);
202         return NULL;
203     }
204
205     if (size == -1)

```



```

206         size = LSPLIST_INITSIZE;
207
208     if ((ptr = calloc(size, sizeof(DBLabelSwitchedPath*))) == NULL)
209     {
210         addError(CRITICAL, "Critical lack of memory in %s at line %d",
211                 __FILE__, __LINE__);
212         free(list);
213         return NULL;
214     }
215
216     list->size = size;
217     list->top = 0;
218     list->cont = ptr;
219
220     return list;
221 }

```

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`.

```

379 {
380     int a,b,index;
381
382     if (list == NULL || list->cont == NULL || lsp == NULL)
383     {
384         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
385                 __FILE__, __LINE__);
386         return -1;
387     }
388
389     // find the position in the list
390     a = 0;
391     b = list->top-1;
392
393     // empty list
394     if (list->top == 0)
395     {
396         addError(WARNING, "Removing inexistent LSP in %s at line %d",
397                 __FILE__, __LINE__);
398         return -1;
399     }
400
401     while (b - a > 1)
402     {
403         int mid = (a + b)/2;
404         int ret = DBLspCompare(lsp, list->cont[mid]);
405
406         if (ret == 1)
407             b = mid;
408         else if (ret == -1)
409             a = mid;
410         else // if (ret == 0)
411         {
412             a = mid;
413             b = mid;
414         }
415     }
416
417     if (DBLspCompare(lsp, list->cont[a]) == 0)
418     {

```

```

419     index = a;
420 }
421 else if (DBlspCompare(lsp, list->cont[b]) == 0)
422 {
423     index = b;
424 }
425 else // not found
426 {
427     addError(WARNING, "Removing inexistent LSP in %s at line %d",
428             __FILE__, __LINE__);
429     return -1;
430 }
431
432 // now delete index
433 memmove(list->cont + index, list->cont + index + 1, (list->top - index - 1)*sizeof(void*));
434 list->top--;
435
436 return 0;
437 }

```

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.

```

20 {
21     DBLabelSwitchedPath* lsp;
22
23     if ((lsp=calloc(1,sizeof(DBLabelSwitchedPath)))==NULL)
24     {
25         addError(CRITICAL, "Critical lack of memory in %s at line %d",
26                 __FILE__, __LINE__);
27         return NULL;
28     }
29
30     if (longListInit(&(lsp->forbidLinks), -1) < 0)
31     {
32         free(lsp);
33         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
34                 __FILE__, __LINE__);
35         return NULL;
36     }
37
38     if (longListInit(&(lsp->path), -1) < 0)
39     {
40         longListEnd(&(lsp->forbidLinks));
41         free(lsp);
42         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
43                 __FILE__, __LINE__);
44         return NULL;
45     }
46
47     if (longListInit(&(lsp->primPath), -1) < 0)
48     {
49         longListEnd(&(lsp->path));
50         longListEnd(&(lsp->forbidLinks));
51         free(lsp);
52         addError(CRITICAL, "Unable to create label switched path in %s at line %d",
53                 __FILE__, __LINE__);
54         return NULL;
55     }

```

```

56
57     if (longListInit(&(lsp->backLSPIDs),-1)<0)
58     {
59         longListEnd(&(lsp->primPath));
60         longListEnd(&(lsp->path));
61         longListEnd(&(lsp->forbidLinks));
62         free(lsp);
63         addError(CRITICAL,"Unable to create label switched path in %s at line %d",
64                 __FILE__,__LINE__);
65         return NULL;
66     }
67
68     lsp->noContentionId=-1; //very important
69
70     return lsp;
71 }

```

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`.

```

1281 {
1282     DataBase *dataBase=NULL;
1283
1284     if ((dataBase=calloc(1,sizeof(DataBase)))==NULL)
1285     {
1286         addError(CRITICAL,"Critical lack of memory in %s at line %d",
1287                 __FILE__,__LINE__);
1288         return NULL;
1289     }
1290
1291     dataBase->id=ID;
1292
1293     if (DBnodeVecInit(&(dataBase->nodeVec),-1)<0)
1294     {
1295         addError(CRITICAL,"Unable to initialize the general node container in %s at line %d",
1296                 __FILE__,__LINE__);
1297         free(dataBase);
1298         return NULL;
1299     }
1300
1301     if (DBlspVecInit(&(dataBase->lspVec),-1)<0)
1302     {
1303         addError(CRITICAL,"Unable to initialize the general LSP container in %s at line %d",
1304                 __FILE__,__LINE__);
1305         DBnodeVecEnd(&(dataBase->nodeVec));
1306         free(dataBase);
1307         return NULL;
1308     }
1309
1310     if (DBlinkTabInit(&(dataBase->linkTab),-1)<0)
1311     {
1312         addError(CRITICAL,"Unable to initialize the general link container in %s at line %d",
1313                 __FILE__,__LINE__);
1314         DBnodeVecEnd(&(dataBase->nodeVec));
1315         DBlspVecEnd(&(dataBase->lspVec));
1316         free(dataBase);
1317         return NULL;
1318     }

```

```

1319
1320     if (longVecInit(&(dataBase->linkSrcVec),LINKTAB_INITSIZE)<0)
1321     {
1322         addError(CRITICAL,"Unable to initialize the link id-src translator in %s at line %d",
1323             __FILE__,__LINE__);
1324         DBnodeVecEnd(&(dataBase->nodeVec));
1325         DBlspVecEnd(&(dataBase->lspVec));
1326         DBlinkTabEnd(&(dataBase->linkTab));
1327         free(dataBase);
1328         return NULL;
1329     }
1330
1331     if (longVecInit(&(dataBase->linkDstVec),LINKTAB_INITSIZE)<0)
1332     {
1333         addError(CRITICAL,"Unable to initialize the link id-dst translator in %s at line %d",
1334             __FILE__,__LINE__);
1335         DBnodeVecEnd(&(dataBase->nodeVec));
1336         DBlspVecEnd(&(dataBase->lspVec));
1337         DBlinkTabEnd(&(dataBase->linkTab));
1338         longVecEnd(&(dataBase->linkSrcVec));
1339         free(dataBase);
1340         return NULL;
1341     }
1342
1343     dataBase->nbNodes=0;
1344     dataBase->nbLinks=0;
1345
1346     return dataBase;
1347 }

```

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.

```

2254 {
2255     long i,j;
2256
2257     printf("Printing info about nodes ...\n");
2258     printf("-----\n");
2259
2260     for (i=0; i<db->nodeVec.size; i++)
2261     {
2262         if (db->nodeVec.cont[i])
2263         {
2264             printf("Node id : %ld\n", i);
2265             printf("-----\n");
2266             DBprintNode(db->nodeVec.cont[i]);
2267         }
2268     }
2269
2270     printf("\nPrinting info about links ...\n");
2271     printf("-----\n");
2272
2273     for (i=0; i<db->linkTab.size; i++)
2274         for (j=0; j<db->linkTab.size; j++)
2275         {
2276             if (db->linkTab.cont[i][j])
2277             {
2278                 printf("Link %ld-%ld (id = %ld)\n", i, j, DBgetLinkID(db, i, j));
2279                 printf("-----\n");
2280

```

```

2281             DBprintLink(db->linkTab.cont[i][j]);
2282         }
2283     }
2284 }
2285 }

```

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().

```

1636 {
1637     int id,ret=0;
1638
1639     if (dataBase == NULL)
1640     {
1641         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1642             __FILE__,__LINE__);
1643         return -1;
1644     }
1645
1646     if ((DBnodeVecGet(&(dataBase->nodeVec),src)==NULL) ||
1647         (DBnodeVecGet(&(dataBase->nodeVec),dst)==NULL) ||
1648         (DBlinkTabGet(&(dataBase->linkTab),src,dst)==NULL))
1649     {
1650         addError(CRITICAL,"Link doesn't exist or database unconsistency in %s at line %d",
1651             __FILE__,__LINE__);
1652         return -1;
1653     }
1654
1655     ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[src]->outNeighb),dst));
1656     ANDERROR(ret,longListRemove(&(dataBase->nodeVec.cont[dst]->inNeighb),src));
1657
1658     ANDERROR(ret,DBlinkTabRemove(&(dataBase->linkTab),src,dst));
1659
1660     id=DBgetLinkID(dataBase,src,dst);
1661     ANDERROR(ret,longVecSet(&(dataBase->linkSrcVec),id,0));
1662     ANDERROR(ret,longVecSet(&(dataBase->linkDstVec),id,0));
1663
1664     while (dataBase->linkSrcVec.cont[dataBase->linkSrcVec.top-1] == 0)
1665         dataBase->linkSrcVec.top--;
1666
1667     if (ret<0)
1668     {
1669         addError(CRITICAL,"Link removal uncomplete in %s at line %d",
1670             __FILE__,__LINE__);
1671     }
1672
1673     dataBase->nbLinks--;
1674
1675     return ret;
1676 }

```

4.13.1.39 int DBremoveLSP (DataBase *, long)

Definition at line 2005 of file database-oli.c.

References addError(), ANDERROR, and DBlinkTabGet.

```

2006 {
2007     DBLabelSwitchedPath *lsp=NULL, *contentLSP=NULL;
2008     int i,ret=0;
2009     DBLink *lnk=NULL;
2010     LongVec isProcessed;
2011
2012     if (dataBase == NULL)
2013     {
2014         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2015             __FILE__,__LINE__);
2016         return -1;
2017     }
2018
2019     if ((lsp = DBlspVecGet(&(dataBase->lspVec), id)) == NULL)
2020     {
2021         addError(CRITICAL,"Trying to remove inexistent LSP (id = %ld) in %s at line %d",
2022             id,__FILE__,__LINE__);
2023         return -1;
2024     }
2025
2026     if (longVecInit(&(isProcessed), dataBase->linkSrcVec.size)<0)
2027     {
2028         addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
2029             __FILE__,__LINE__);
2030         return -1;
2031     }
2032
2033     #if defined SIMULATOR
2034         // Remove the LSP from each link list and update all the linkstates
2035         for (i=0;i<lsp->path.top-1;i++)
2036         {
2037             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
2038                 lsp->path.cont[i+1]);
2039             ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
2040             ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->path.cont[i],
2041                 lsp->path.cont[i+1], &(lnk->state), lsp));
2042             isProcessed.cont[lnk->id] = 1;
2043         }
2044         if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
2045         {
2046             for (i=0;i<lsp->primPath.top-1;i++)
2047             {
2048                 lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
2049                     lsp->primPath.cont[i+1]);
2050                 if (isProcessed.cont[lnk->id] == 0)
2051                 {
2052                     ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->primPath.cont[i],
2053                         lsp->primPath.cont[i+1], &(lnk->state), lsp));
2054                     isProcessed.cont[lnk->id] = 1;
2055                 }
2056             }
2057         }
2058     #elif defined AGENT
2059         // Remove the LSP to the link attached to the agent and update the linkstate
2060         for (i=0;i<lsp->path.top-1;i++)
2061         {
2062             lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->path.cont[i],
2063                 lsp->path.cont[i+1]);
2064             ANDERROR(ret,DBlspListRemove(&(lnk->lspList),lsp));
2065
2066             if (lsp->path.cont[i] == dataBase->id)
2067             {
2068                 ANDERROR(ret,DBupdateLSONRemove(dataBase, lsp->path.cont[i],
2069                     lsp->path.cont[i+1], &(lnk->state), lsp));
2070                 isProcessed.cont[lnk->id] = 1;

```

```

2071     }
2072 }
2073 if ((lsp->type == GLOBAL_BACK) || (lsp->type == LOCAL_BACK))
2074 {
2075     for (i=0;i<lsp->primPath.top-1;i++)
2076     {
2077         lnk=DBlinkTabGet(&(dataBase->linkTab),lsp->primPath.cont[i],
2078                         lsp->primPath.cont[i+1]);
2079
2080         if (lsp->primPath.cont[i] == dataBase->id)
2081         {
2082             if (isProcessed.cont[lnk->id] == 0)
2083             {
2084                 ANDERROR(ret,DBupdateLSPonRemove(dataBase, lsp->primPath.cont[i],
2085                                                    lsp->primPath.cont[i+1], &(lnk->state), lsp));
2086             }
2087             break;
2088         }
2089     }
2090 }
2091 #else
2092     // Generate an error;
2093     COMPILE_ERROR;
2094 #endif
2095
2096     longVecEnd(&(isProcessed));
2097
2098     // remove the lsp from the global list
2099     ANDERROR(ret,DBlspVecRemove(&(dataBase->lspVec), id));
2100
2101     if (lsp->noContentionId>=0)
2102     {
2103         if ((contentLSP=DBlspVecGet(&(dataBase->lspVec),lsp->noContentionId))==NULL)
2104         {
2105             addError(WARNING,"Unable to get no contention LSP in %s at line %d",
2106                     __FILE__,__LINE__);
2107             // not critical enough to abort
2108         }
2109         contentLSP->noContentionId=-1;
2110     }
2111
2112     // free the lsp
2113     DBlspDestroy(lsp);
2114
2115     if (ret<0)
2116     {
2117         addError(CRITICAL,"LSP removal uncomplete in %s at line %d",
2118                 __FILE__,__LINE__);
2119     }
2120
2121     return ret;
2122 }

```

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`.

```

1500 {
1501     DBNode *node=NULL;
1502     int ret=0;

```

```

1503
1504     if (dataBase == NULL)
1505     {
1506         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1507             __FILE__,__LINE__);
1508         return -1;
1509     }
1510
1511     if ((node=DBnodeVecGet(&(dataBase->nodeVec),id)) == NULL)
1512     {
1513         addError(CRITICAL,"Trying to remove an inexistent node in %s at line %d",
1514             __FILE__,__LINE__);
1515         return -1;
1516     }
1517
1518     // remember that DBremoveLink will update the neighbour list
1519     while(node->inNeighb.top > 0)
1520     {
1521         ANDERROR(ret,DBremoveLink(dataBase,node->inNeighb.cont[node->inNeighb.top-1],id));
1522     }
1523
1524     // remember that DBremoveLink will update the neighbour list
1525     while(node->outNeighb.top > 0)
1526     {
1527         ANDERROR(ret,DBremoveLink(dataBase,id,node->outNeighb.cont[node->outNeighb.top-1]));
1528     }
1529
1530     ANDERROR(ret,DBnodeVecRemove(&(dataBase->nodeVec),id));
1531
1532     if (ret<0)
1533     {
1534         addError(CRITICAL,"Node removal uncomplete in %s at line %d",
1535             __FILE__,__LINE__);
1536     }
1537
1538     dataBase->nbLinks--;
1539
1540     return ret;
1541 }

```

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.

```

2181 {
2182     DBLink *lnk=NULL;
2183
2184     if (dataBase == NULL || newLS == NULL)
2185     {
2186         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
2187             __FILE__,__LINE__);
2188         return -1;
2189     }
2190
2191     if ((lnk=DBlinkTabGet(&(dataBase->linkTab),src,dst)) == NULL)
2192     {
2193         addError(CRITICAL,"Inexistent Link (src = %ld, dst = %ld) in %s at line %d",
2194             src,dst,__FILE__,__LINE__);
2195         return -1;
2196     }
2197

```



```
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 }
```

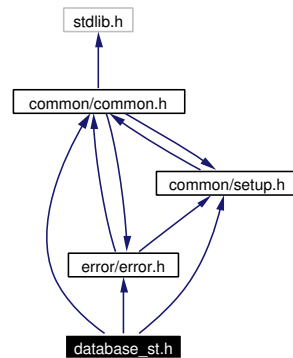
4.14 database_st.h File Reference

```
#include "common/common.h"
```

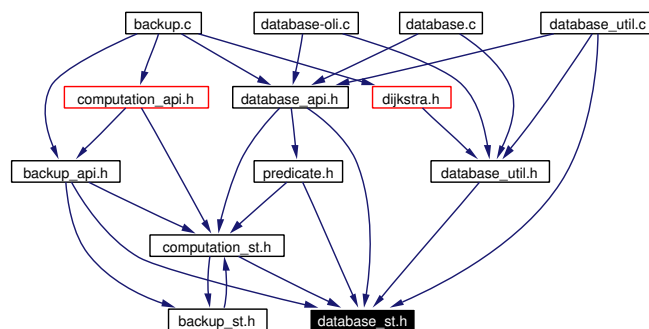
```
#include "common/setup.h"
```

```
#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](#)

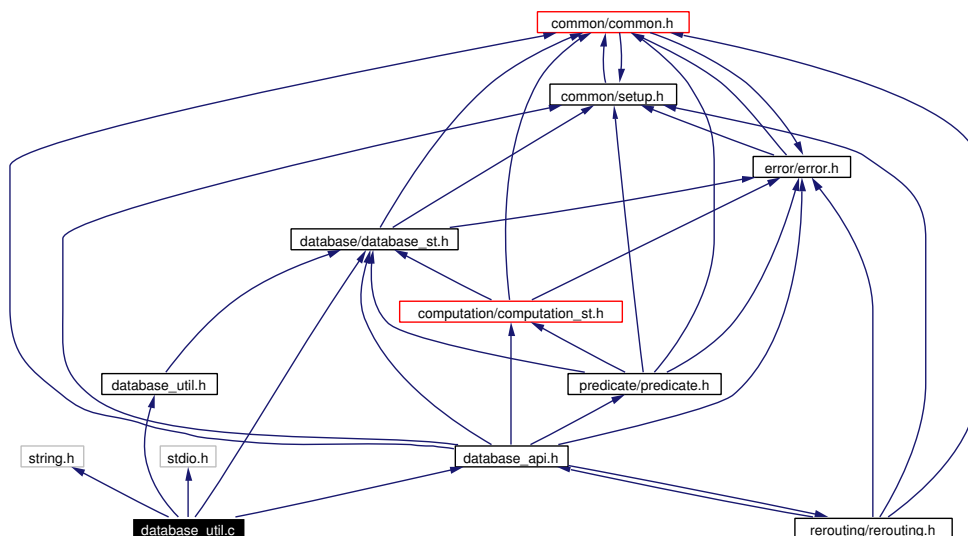
Definition at line 17 of file database_st.h.

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 already allocated somewhere else.
- **int DBnodeDestroy (DBNode *node)**
clear function to free the resources of a Link object allocated on the heap.
- **int DBnodeEnd (DBNode *node)**
clear function to free the resources 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 already 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 [DBlinkTabResize\(\)](#).

```

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));

```

```

215     free(link);
216
217     return 0;
218 }

```

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.

```

222 {
223     if (link == NULL)
224     {
225         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
226                 __FILE__, __LINE__);
227         return -1;
228     }
229
230     DBlinkStateEnd(&link->state);
231     DBlspListEnd(&(link->lspList));
232
233     return 0;
234 }

```

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.

```

177 {
178     if (link == NULL)
179     {
180         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
181                 __FILE__, __LINE__);
182         return -1;
183     }
184
185     if (DBlinkStateInit(&(link->state)) == -1)
186     {
187         addError(CRITICAL, "Error while initializing LinkState in %s at line %d",
188                 __FILE__, __LINE__);
189         return -1;
190     }
191
192     if (DBlspListInit(&(link->lspList), -1) < 0)
193     {
194         addError(CRITICAL, "Error while initializing LinkState in %s at line %d",
195                 __FILE__, __LINE__);
196         DBlinkStateEnd(&(link->state));
197         return -1;
198     }
199
200     return 0;
201 }

```

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().

```

145 {
146     DBLink* ptr=NULL;
147
148     if ((ptr = calloc(1,sizeof(DBLink))) == NULL)
149     {
150         addError(CRITICAL,"Critical lack of memory in %s at line %d",
151             __FILE__,__LINE__);
152         return NULL;
153     }
154
155     if (DBlinkStateInit(&(ptr->state)) == -1)
156     {
157         addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
158             __FILE__,__LINE__);
159         free(ptr);
160         return NULL;
161     }
162
163     if (DBlspListInit(&(ptr->lspList),-1) < 0)
164     {
165         addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
166             __FILE__,__LINE__);
167         DBlinkStateEnd(&(ptr->state));
168         free(ptr);
169         return NULL;
170     }
171
172     return ptr;
173 }
```

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.

```

800 {
801     int i,j;
802
803     if (tab == NULL || tab->cont == NULL)
804     {
805         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
806             __FILE__,__LINE__);
807         return -1;
808     }
809
810     for (i=0;i<tab->size;i++)
811     {
812         for (j=0; j<tab->size; j++)
813         {
814             if (tab->cont[i][j] != NULL)
815             {
```



```

816         DBlinkDestroy(tab->cont[i][j]);
817     }
818 }
819 free(tab->cont[i]);
820 }
821 free(tab->cont);
822 free(tab);
823
824 return 0;
825 }

```

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\(\)](#).

```

828 {
829     int i,j;
830
831     if (tab == NULL || tab->cont == NULL)
832     {
833         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
834             __FILE__,__LINE__);
835         return -1;
836     }
837
838     for (i=0;i<tab->size;i++)
839     {
840         for (j=0; j<tab->size; j++)
841         {
842             if (tab->cont[i][j] != NULL)
843             {
844                 DBlinkDestroy(tab->cont[i][j]);
845             }
846         }
847         free(tab->cont[i]);
848     }
849
850     free(tab->cont);
851     tab->cont = NULL;
852     tab->size = 0;
853
854     return 0;
855 }

```

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 [DBLinkTab_::size](#).

Referenced by [DBnew\(\)](#).

```

757 {
758     DBLink ***ptr=NULL;
759     int i;
760

```

```

761     if (tab == NULL)
762     {
763         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
764             __FILE__,__LINE__);
765         return -1;
766     }
767
768     if (size == -1)
769         size = LINKTAB_INITSIZE;
770
771     if ((ptr=(calloc(size,sizeof(DBLink**)))==NULL)
772     {
773         addError(CRITICAL,"Critical lack of memory in %s at line %d",
774             __FILE__,__LINE__);
775         return -1;
776     }
777     else
778     {
779         for (i=0;i<size;i++)
780         {
781             if ((ptr[i]=(calloc(size, sizeof(DBLink*)))==NULL)
782             {
783                 addError(CRITICAL,"Critical lack of memory in %s at line %d",
784                     __FILE__,__LINE__);
785                 for (i=i-1;i>=0;i--)
786                     free(ptr[i]);
787                 free(ptr);
788                 return -1;
789             }
790         }
791     }
792
793     tab->size=size;
794     tab->cont=ptr;
795
796     return 0;
797 }

```

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.

```

711 {
712     DBLinkTab *tab=NULL;
713     DBLink ***ptr=NULL;
714     int i;
715
716     if ((tab = calloc(1,sizeof(DBLinkTab))) == NULL)
717     {
718         addError(CRITICAL,"Critical lack of memory in %s at line %d",
719             __FILE__,__LINE__);
720         return NULL;
721     }
722
723     if (size == -1)
724         size = LINKTAB_INITSIZE;
725
726     if ((ptr=(calloc(size,sizeof(DBLink**)))==NULL)
727     {
728         addError(CRITICAL,"Critical lack of memory in %s at line %d",
729             __FILE__,__LINE__);

```

```

730     free(tab);
731     return NULL;
732 }
733 else
734 {
735     for (i=0;i<size;i++)
736     {
737         if ((ptr[i]=(calloc(size, sizeof(DBLink*))))==NULL)
738         {
739             addError(CRITICAL,"Critical lack of memory in %s at line %d",
740                     __FILE__,__LINE__);
741             for (i=i-1;i>=0;i--)
742                 free(ptr[i]);
743             free(ptr);
744             free(tab);
745             return NULL;
746         }
747     }
748 }
749
750 tab->size=size;
751 tab->cont=ptr;
752
753 return tab;
754 }

```

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().

```

964 {
965     if (tab == NULL || tab->cont == NULL ||
966         src < 0 || dst < 0 || src >= tab->size || dst >= tab->size ||
967         tab->cont[src][dst] == NULL)
968     {
969         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
970                 __FILE__,__LINE__);
971         return -1;
972     }
973
974     DBlinkDestroy(tab->cont[src][dst]);
975     tab->cont[src][dst]=NULL;
976
977     return 0;
978 }

```

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().

```

859 {
860     DBLink*** ptr=NULL;

```

```

861     DBLink** ptr2=NULL;
862     int i,j;
863
864     if (tab == NULL || tab->cont == NULL)
865     {
866         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
867             __FILE__,__LINE__);
868         return -1;
869     }
870
871     if (size < tab->size)
872     {
873         for (i=size;i<tab->size;i++)
874         {
875             for (j=size; j<tab->size; j++)
876             {
877                 if (tab->cont[i][j] != NULL)
878                 {
879                     DBlinkDestroy(tab->cont[i][j]);
880                     tab->cont[i][j]=NULL;
881                 }
882             }
883         }
884         free(tab->cont[i]);
885     }
886
887     if ((ptr = realloc(tab->cont,size * sizeof(DBLink**))) == NULL)
888     {
889         addError(CRITICAL,"Critical lack of memory in %s at line %d",
890             __FILE__,__LINE__);
891         return -1;
892     }
893     else
894     {
895         tab->cont = ptr;
896
897         for (i=0;i<min(tab->size,size);i++)
898         {
899             if ((ptr2 = realloc(ptr[i], size * sizeof(DBLink*)))==NULL)
900             {
901                 addError(CRITICAL,"Critical lack of memory in %s at line %d",
902                     __FILE__,__LINE__);
903                 tab->size=min(tab->size,size);
904                 return -1;
905             }
906
907             ptr[i] = ptr2;
908
909             if (size > tab->size)
910             {
911                 memset(ptr2 + tab->size, 0, (size-tab->size) * sizeof(DBLink*));
912             }
913         }
914
915         if (size > tab->size)
916         {
917             for (i=tab->size;i<size;i++)
918             {
919                 if ((ptr[i] = calloc(size, sizeof(DBLink*)))==NULL)
920                 {
921                     addError(CRITICAL,"Critical lack of memory in %s at line %d",
922                         __FILE__,__LINE__);
923                     tab->size=i;
924                     return -1;
925                 }
926             }
927         }

```

```

928     }
929
930     tab->size=size;
931
932     return 0;
933 }

```

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()`.

```

936 {
937     long resize;
938
939     if (tab == NULL || tab->cont == NULL || src < 0 || dst < 0)
940     {
941         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
942                 __FILE__, __LINE__);
943         return -1;
944     }
945
946     resize=max(src,dst)+1;
947
948     if (resize > tab->size)
949     {
950         if (DBlinkTabResize(tab,max(2*tab->size,resize))<0)
951         {
952             addError(CRITICAL, "Unable to resize link table prior to insertion in %s at line %d",
953                     __FILE__, __LINE__);
954             return -1;
955         }
956     }
957
958     tab->cont[src][dst]=lnk;
959
960     return 0;
961 }

```

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`.

```

566 {
567     int i;
568
569     if (vec == NULL || vec->cont == NULL)
570     {
571         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
572                 __FILE__, __LINE__);
573         return -1;
574     }
575
576     for (i=0; i<vec->size; i++)
577     {
578         if (vec->cont[i]!=NULL)

```

```

579     {
580         DBlspDestroy(vec->cont[i]);
581     }
582 }
583
584 free(vec->cont);
585 free(vec);
586
587 return 0;
588 }

```

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\(\)](#).

```

591 {
592     int i;
593
594     if (vec == NULL || vec->cont == NULL)
595     {
596         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
597             __FILE__, __LINE__);
598         return -1;
599     }
600
601     for (i=0; i<vec->size; i++)
602     {
603         if (vec->cont[i]!=NULL)
604         {
605             DBlspDestroy(vec->cont[i]);
606         }
607     }
608
609     free(vec->cont);
610     vec->cont = NULL;
611     vec->size = 0;
612
613     return 0;
614 }

```

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\(\)](#).

```

539 {
540     void* ptr=NULL;
541
542     if (vec == NULL)
543     {
544         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
545             __FILE__, __LINE__);
546         return -1;

```

```

547     }
548
549     if (size == -1)
550         size = LSPVEC_INITSIZE;
551
552     if ((ptr = calloc(size, sizeof(DBLabelSwitchedPath*))) == NULL)
553     {
554         addError(CRITICAL, "Critical lack of memory in %s at line %d",
555                 __FILE__, __LINE__);
556         return -1;
557     }
558
559     vec->size = size;
560     vec->cont = ptr;
561
562     return 0;
563 }

```

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`.

```

510 {
511     DBLSPVec *vec=NULL;
512     void* ptr=NULL;
513
514     if ((vec = calloc(1, sizeof(DBLSPVec))) == NULL)
515     {
516         addError(CRITICAL, "Critical lack of memory in %s at line %d",
517                 __FILE__, __LINE__);
518         return NULL;
519     }
520
521     if (size == -1)
522         size = LSPVEC_INITSIZE;
523
524     if ((ptr = calloc(size, sizeof(DBLabelSwitchedPath*))) == NULL)
525     {
526         addError(CRITICAL, "Critical lack of memory in %s at line %d",
527                 __FILE__, __LINE__);
528         free(vec);
529         return NULL;
530     }
531
532     vec->size = size;
533     vec->cont = ptr;
534
535     return vec;
536 }

```

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`.

```

690 {
691     if (vec == NULL || vec->cont == NULL ||

```

```

692     id < 0 || id >= vec->size || vec->cont[id] == NULL)
693 {
694     addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
695             __FILE__, __LINE__);
696     return -1;
697 }
698
699 vec->cont[id]=NULL;
700
701 return 0;
702 }

```

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().

```

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(DBLabelSwitchedPath*));
650     }
651
652     vec->size=size;
653     vec->cont=ptr;
654
655     return 0;
656 }

```


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().

```

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__);
664         return -1;
665     }
666
667     if (id >= vec->size)
668     {
669         if (DBLSPVecResize(vec, max(2*vec->size, id+1)) < 0)
670         {
671             addError(CRITICAL, "Unable to resize LSP vector prior to insertion in %s at line %d",
672                 __FILE__, __LINE__);
673             return -1;
674         }
675     }
676
677     if (vec->cont[id] != NULL)
678     {
679         addError(CRITICAL, "Trying to add an LSP with a reserved ID in %s at line %d",
680             __FILE__, __LINE__);
681         return -1;
682     }
683
684     vec->cont[id] = lsp;
685
686     return 0;
687 }
```

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().

```

78 {
79     if (node == NULL)
80     {
81         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
82             __FILE__, __LINE__);
83         return -1;
84     }
85
86     longListEnd(&(node->inNeighb));
87     longListEnd(&(node->outNeighb));
88     free(node);
89
90     return 0;
91 }
```

4.15.1.20 int DBnodeEnd (DBNode * node)

clear function to free the resources 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.

```

95 {
96     if (node == NULL)
97     {
98         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
99             __FILE__, __LINE__);
100         return -1;
101     }
102
103     longListEnd(&(amp;node->inNeighb));
104     longListEnd(&(amp;node->outNeighb));
105
106     return 0;
107 }
```

4.15.1.21 int DBnodeInit (DBNode * node)

initialize a DBNode object already allocated somewhere else.

Definition at line 49 of file database_util.c.

References addError(), CRITICAL, DBNode::inNeighb, longListEnd, longListInit, and DBNode::outNeighb.

```

50 {
51     if (node == NULL)
52     {
53         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
54             __FILE__, __LINE__);
55         return -1;
56     }
57
58     if (longListInit(&(amp;node->inNeighb), -1) < 0)
59     {
60         addError(CRITICAL, "Unable to initialize the incoming neighbour list in %s at line %d",
61             __FILE__, __LINE__);
62         return -1;
63     }
64
65     if (longListInit(&(amp;node->outNeighb), -1) < 0)
66     {
67         addError(CRITICAL, "Unable to initialize the outgoing neighbour list in %s at line %d",
68             __FILE__, __LINE__);
69         longListEnd(&(amp;node->inNeighb));
70         return -1;
71     }
72
73     return 0;
74 }
```

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()`.

```

19 {
20     DBNode* ptr=NULL;
21
22     if ((ptr = calloc(1,sizeof(DBNode))) == NULL)
23     {
24         addError(CRITICAL,"Critical lack of memory in %s at line %d",
25             __FILE__,__LINE__);
26         return NULL;
27     }
28
29     if (longListInit(&(ptr->inNeighb),-1) < 0)
30     {
31         addError(CRITICAL,"Unable to initialize the incoming neighbour list in %s at line %d",
32             __FILE__,__LINE__);
33         return NULL;
34     }
35
36     if (longListInit(&(ptr->outNeighb),-1) < 0)
37     {
38         addError(CRITICAL,"Unable to initialize the outgoing neighbour list in %s at line %d",
39             __FILE__,__LINE__);
40         longListEnd(&(ptr->inNeighb));
41         free(ptr);
42         return NULL;
43     }
44
45     return ptr;
46 }
```

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`.

```

350 {
351     int i;
352
353     if (vec == NULL || vec->cont == NULL)
354     {
355         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
356             __FILE__,__LINE__);
357         return -1;
358     }
359
360     for (i=0; i<vec->size; i++)
361     {
362         if (vec->cont[i]!=NULL)
363         {
364             DBnodeDestroy(vec->cont[i]);
365         }
366     }
367
368     free(vec->cont);
369     free(vec);
370
371     return 0;
372 }
```

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().

```

375 {
376     int i;
377
378     if (vec == NULL || vec->cont == NULL)
379     {
380         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
381                 __FILE__, __LINE__);
382         return -1;
383     }
384
385     for (i=0; i<vec->size; i++)
386     {
387         if (vec->cont[i]!=NULL)
388         {
389             DBnodeDestroy(vec->cont[i]);
390         }
391     }
392
393     free(vec->cont);
394     vec->cont = NULL;
395     vec->size = 0;
396     vec->top = 0;
397
398     return 0;
399 }
```

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().

```

322 {
323     void* ptr=NULL;
324
325     if (vec == NULL)
326     {
327         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
328                 __FILE__, __LINE__);
329         return -1;
330     }
331
332     if (size == -1)
333         size = NODEVEC_INITSIZE;
334
335     if ((ptr = calloc(size, sizeof(DBNode*))) == NULL)
336     {
337         addError(CRITICAL, "Critical lack of memory in %s at line %d",
338                 __FILE__, __LINE__);
339         return -1;
340     }
```

```

341
342     vec->size = size;
343     vec->top = 0;
344     vec->cont = ptr;
345
346     return 0;
347 }

```

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`.

```

291 {
292     DBNodeVec *vec=NULL;
293     void* ptr=NULL;
294
295     if ((vec = calloc(1,sizeof(DBNodeVec))) == NULL)
296     {
297         addError(CRITICAL,"Critical lack of memory in %s at line %d",
298             __FILE__,__LINE__);
299         return NULL;
300     }
301
302     if (size == -1)
303         size = NODEVEC_INITSIZE;
304
305     if ((ptr = calloc(size,sizeof(DBNode*))) == NULL)
306     {
307         addError(CRITICAL,"Critical lack of memory in %s at line %d",
308             __FILE__,__LINE__);
309         free(vec);
310         return NULL;
311     }
312
313     vec->size = size;
314     vec->top = 0;
315     vec->cont = ptr;
316
317     return vec;
318 }

```

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()`.

```

486 {
487     if (vec == NULL || vec->cont == NULL ||
488         id < 0 || id >= vec->size || vec->cont[id] == NULL)
489     {
490         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
491             __FILE__,__LINE__);
492         return -1;
493     }

```

```

494
495     DBnodeDestroy(vec->cont[id]);
496     vec->cont[id]=NULL;
497
498     while (vec->cont[vec->top-1] == NULL)
499         vec->top--;
500
501     return 0;
502 }

```

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, DBNodeVec_::size, and DBNodeVec_::top.

Referenced by DBnodeVecSet().

```

402 {
403     void *ptr=NULL;
404     int i;
405
406     if (vec == NULL || vec->cont == NULL)
407     {
408         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
409             __FILE__, __LINE__);
410         return -1;
411     }
412
413     if (size < vec->size)
414     {
415         for (i=size;i<vec->size;i++)
416         {
417             if (vec->cont[i]!=NULL)
418             {
419                 DBnodeDestroy(vec->cont[i]);
420                 vec->cont[i]=NULL;
421             }
422         }
423
424         if (size < vec->top)
425         {
426             vec->top = size;
427             while (vec->cont[vec->top-1] == NULL)
428                 vec->top--;
429         }
430     }
431 }
432
433
434
435 if ((ptr = realloc(vec->cont, size * sizeof(DBNode*))) == NULL)
436 {
437     addError(CRITICAL,"Critical lack of memory in %s at line %d",
438         __FILE__, __LINE__);
439     return -1;
440 }
441
442 if (size > vec->size)
443 {
444     memset(ptr + (vec->size * sizeof(DBNode*)), 0, (size-vec->size) * sizeof(DBNode*));
445 }
446

```

```

447     vec->size=size;
448     vec->cont=ptr;
449
450     return 0;
451 }

```

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()`.

```

454 {
455     if (vec == NULL || vec->cont == NULL || node == NULL || id < 0)
456     {
457         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
458                 __FILE__, __LINE__);
459         return -1;
460     }
461
462     if (id >= vec->size)
463     {
464         if (DBnodeVecResize(vec, max(2*vec->size, id+1)) < 0)
465         {
466             addError(CRITICAL, "Unable to resize node vector prior to insertion in %s at line %d",
467                     __FILE__, __LINE__);
468             return -1;
469         }
470     }
471
472     if (vec->cont[id] != NULL)
473     {
474         addError(CRITICAL, "Trying to add a node with a reserved ID in %s at line %d",
475                 __FILE__, __LINE__);
476         return -1;
477     }
478
479     vec->cont[id]=node;
480     vec->top = max(vec->top, id+1);
481
482     return 0;
483 }

```

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()`.

```

238 {
239     long i, oa;
240     double ptot, rtot;
241     DBLabelSwitchedPath* lsp=NULL;

```

```

242
243     if (link == NULL)
244     {
245         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
246             __FILE__,__LINE__);
247         return;
248     }
249
250     printf("\tList of LSPs\n");
251     printf("\t-----\n");
252
253     for (i=0; i<link->lspList.top; ++i)
254     {
255         lsp = link->lspList.cont[i];
256         printf("%ld ", lsp->id);
257     }
258
259     printf("\n\n");
260
261     printf("\tLink-state\n");
262     printf("\t-----\n");
263
264     for (oa=0; oa<NB_OA; ++oa)
265     {
266         ptot = 0;
267         rtot = 0;
268
269         printf("\tCapacity[%ld] = %f\n", oa, link->state.cap[oa]);
270
271         for (i=0; i<NB_PREEMPTION; ++i)
272         {
273             ptot += link->state.pbw[oa][i];
274             rtot += link->state.rbw[oa][i];
275         }
276
277         printf("\ttpbw[%ld] = %f\n", oa, ptot);
278         printf("\trbw[%ld] = %f\n", oa, rtot);
279     }
280
281     printf("\n\n");
282
283 }

```

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().

```

111 {
112     long i;
113
114     if (node == NULL)
115     {
116         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
117             __FILE__,__LINE__);
118         return;
119     }
120
121     printf("Incoming neighbors : \n");
122

```

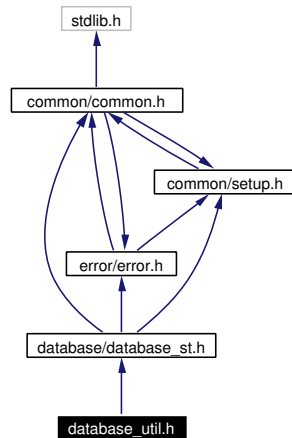


```
123     for (i=0; i<node->inNeighb.top; i++)
124     {
125         printf("%ld ", node->inNeighb.cont[i]);
126     }
127
128     printf("\nOutgoing neighbors : \n");
129
130     for (i=0; i<node->outNeighb.top; i++)
131     {
132         printf("%ld ", node->outNeighb.cont[i]);
133     }
134
135     printf("\n");
136 }
```

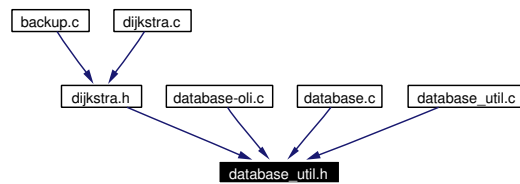
4.16 database_util.h File Reference

```
#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 already allocated somewhere else.
- int [DBnodeDestroy](#) ([DBNode](#) *)
clear function to free the resources 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 already allocated somewhere else.
- int [DBlinkDestroy](#) ([DBLink](#) *)
clear function to free the resources 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 resources 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     {
```

```

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));
215     free(link);
216
217     return 0;
218 }

```

4.16.3.2 int DBlinkEnd ()

4.16.3.3 int DBlinkInit ([DBLink](#) *)

initialize a Link object already allocated somewhere else.

Definition at line 176 of file database_util.c.

References [addError\(\)](#), [CRITICAL](#), [DBlinkStateEnd\(\)](#), [DBlinkStateInit\(\)](#), [DBlspListInit\(\)](#), [DBLink::lspList](#), and [DBLink::state](#).

```

177 {
178     if (link == NULL)
179     {
180         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
181                 __FILE__, __LINE__);
182         return -1;
183     }
184
185     if (DBlinkStateInit(&(link->state)) == -1)
186     {
187         addError(CRITICAL, "Error while initializing LinkState in %s at line %d",
188                 __FILE__, __LINE__);
189         return -1;
190     }
191
192     if (DBlspListInit(&(link->lspList), -1) < 0)
193     {
194         addError(CRITICAL, "Error while initializing LinkState in %s at line %d",
195                 __FILE__, __LINE__);
196         DBlinkStateEnd(&(link->state));
197         return -1;
198     }
199
200     return 0;
201 }

```

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\(\)](#).

```

145 {

```

```

146     DBLink* ptr=NULL;
147
148     if ((ptr = calloc(1,sizeof(DBLink))) == NULL)
149     {
150         addError(CRITICAL,"Critical lack of memory in %s at line %d",
151             __FILE__,__LINE__);
152         return NULL;
153     }
154
155     if (DBlinkStateInit(&(ptr->state)) == -1)
156     {
157         addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
158             __FILE__,__LINE__);
159         free(ptr);
160         return NULL;
161     }
162
163     if (DBlspListInit(&(ptr->lspList),-1) < 0)
164     {
165         addError(CRITICAL,"Error while initializing LinkState in %s at line %d",
166             __FILE__,__LINE__);
167         DBlinkStateEnd(&(ptr->state));
168         free(ptr);
169         return NULL;
170     }
171
172     return ptr;
173 }

```

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.

```

800 {
801     int i,j;
802
803     if (tab == NULL || tab->cont == NULL)
804     {
805         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
806             __FILE__,__LINE__);
807         return -1;
808     }
809
810     for (i=0;i<tab->size;i++)
811     {
812         for (j=0; j<tab->size; j++)
813         {
814             if (tab->cont[i][j] != NULL)
815             {
816                 DBlinkDestroy(tab->cont[i][j]);
817             }
818         }
819         free(tab->cont[i]);
820     }
821     free(tab->cont);
822     free(tab);
823
824     return 0;
825 }

```

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().

```

828 {
829     int i,j;
830
831     if (tab == NULL || tab->cont == NULL)
832     {
833         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
834                 __FILE__,__LINE__);
835         return -1;
836     }
837
838     for (i=0;i<tab->size;i++)
839     {
840         for (j=0; j<tab->size; j++)
841         {
842             if (tab->cont[i][j] != NULL)
843             {
844                 DBlinkDestroy(tab->cont[i][j]);
845             }
846         }
847         free(tab->cont[i]);
848     }
849
850     free(tab->cont);
851     tab->cont = NULL;
852     tab->size = 0;
853
854     return 0;
855 }
```

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().

```

757 {
758     DBLink ***ptr=NULL;
759     int i;
760
761     if (tab == NULL)
762     {
763         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
764                 __FILE__,__LINE__);
765         return -1;
766     }
767
768     if (size == -1)
769         size = LINKTAB_INITSIZE;
770
771     if ((ptr=(calloc(size,sizeof(DBLink**)))==NULL)
772     {
773         addError(CRITICAL,"Critical lack of memory in %s at line %d",
```

```

774         __FILE__, __LINE__);
775     return -1;
776 }
777 else
778 {
779     for (i=0;i<size;i++)
780     {
781         if ((ptr[i]=(calloc(size, sizeof(DBLink*))))==NULL)
782         {
783             addError(CRITICAL,"Critical lack of memory in %s at line %d",
784                 __FILE__, __LINE__);
785             for (i=i-1;i>=0;i--)
786                 free(ptr[i]);
787             free(ptr);
788             return -1;
789         }
790     }
791 }
792
793 tab->size=size;
794 tab->cont=ptr;
795
796 return 0;
797 }

```

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.

```

711 {
712     DBLinkTab *tab=NULL;
713     DBLink ***ptr=NULL;
714     int i;
715
716     if ((tab = calloc(1,sizeof(DBLinkTab))) == NULL)
717     {
718         addError(CRITICAL,"Critical lack of memory in %s at line %d",
719             __FILE__, __LINE__);
720         return NULL;
721     }
722
723     if (size == -1)
724         size = LINKTAB_INITSIZE;
725
726     if ((ptr=(calloc(size,sizeof(DBLink*))))==NULL)
727     {
728         addError(CRITICAL,"Critical lack of memory in %s at line %d",
729             __FILE__, __LINE__);
730         free(tab);
731         return NULL;
732     }
733     else
734     {
735         for (i=0;i<size;i++)
736         {
737             if ((ptr[i]=(calloc(size, sizeof(DBLink*))))==NULL)
738             {
739                 addError(CRITICAL,"Critical lack of memory in %s at line %d",
740                     __FILE__, __LINE__);
741                 for (i=i-1;i>=0;i--)
742                     free(ptr[i]);

```



```

743         free(ptr);
744         free(tab);
745         return NULL;
746     }
747 }
748 }
749
750     tab->size=size;
751     tab->cont=ptr;
752
753     return tab;
754 }
```

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()`.

```

964 {
965     if (tab == NULL || tab->cont == NULL ||
966         src < 0 || dst < 0 || src >= tab->size || dst >= tab->size ||
967         tab->cont[src][dst] == NULL)
968     {
969         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
970             __FILE__, __LINE__);
971         return -1;
972     }
973
974     DBlinkDestroy(tab->cont[src][dst]);
975     tab->cont[src][dst]=NULL;
976
977     return 0;
978 }
```

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()`.

```

859 {
860     DBLink*** ptr=NULL;
861     DBLink** ptr2=NULL;
862     int i,j;
863
864     if (tab == NULL || tab->cont == NULL)
865     {
866         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
867             __FILE__, __LINE__);
868         return -1;
869     }
870
871     if (size < tab->size)
872     {
873         for (i=size;i<tab->size;i++)
```

```

874     {
875         for (j=size; j<tab->size; j++)
876         {
877             if (tab->cont[i][j] != NULL)
878             {
879                 DBlinkDestroy(tab->cont[i][j]);
880                 tab->cont[i][j]=NULL;
881             }
882         }
883     }
884     free(tab->cont[i]);
885 }
886
887 if ((ptr = realloc(tab->cont,size * sizeof(DBLink**))) == NULL)
888 {
889     addError(CRITICAL,"Critical lack of memory in %s at line %d",
890             __FILE__,__LINE__);
891     return -1;
892 }
893 else
894 {
895     tab->cont = ptr;
896
897     for (i=0;i<min(tab->size,size);i++)
898     {
899         if ((ptr2 = realloc(ptr[i], size * sizeof(DBLink*)))==NULL)
900         {
901             addError(CRITICAL,"Critical lack of memory in %s at line %d",
902                     __FILE__,__LINE__);
903             tab->size=min(tab->size,size);
904             return -1;
905         }
906
907         ptr[i] = ptr2;
908
909         if (size > tab->size)
910         {
911             memset(ptr2 + tab->size, 0, (size-tab->size) * sizeof(DBLink*));
912         }
913     }
914
915     if (size > tab->size)
916     {
917         for (i=tab->size;i<size;i++)
918         {
919             if ((ptr[i] = calloc(size, sizeof(DBLink*)))==NULL)
920             {
921                 addError(CRITICAL,"Critical lack of memory in %s at line %d",
922                         __FILE__,__LINE__);
923                 tab->size=i;
924                 return -1;
925             }
926         }
927     }
928 }
929
930 tab->size=size;
931
932 return 0;
933 }

```

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().

```

936 {
937     long resize;
938
939     if (tab == NULL || tab->cont == NULL || src < 0 || dst < 0)
940     {
941         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
942                 __FILE__, __LINE__);
943         return -1;
944     }
945
946     resize = max(src, dst) + 1;
947
948     if (resize > tab->size)
949     {
950         if (DBlinkTabResize(tab, max(2 * tab->size, resize)) < 0)
951         {
952             addError(CRITICAL, "Unable to resize link table prior to insertion in %s at line %d",
953                     __FILE__, __LINE__);
954             return -1;
955         }
956     }
957
958     tab->cont[src][dst] = lnk;
959
960     return 0;
961 }

```

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.

```

566 {
567     int i;
568
569     if (vec == NULL || vec->cont == NULL)
570     {
571         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
572                 __FILE__, __LINE__);
573         return -1;
574     }
575
576     for (i = 0; i < vec->size; i++)
577     {
578         if (vec->cont[i] != NULL)
579         {
580             DBlspDestroy(vec->cont[i]);
581         }
582     }
583
584     free(vec->cont);
585     free(vec);
586
587     return 0;
588 }

```

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().

```

591 {
592     int i;
593
594     if (vec == NULL || vec->cont == NULL)
595     {
596         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
597                 __FILE__, __LINE__);
598         return -1;
599     }
600
601     for (i=0; i<vec->size; i++)
602     {
603         if (vec->cont[i]!=NULL)
604         {
605             DBlspDestroy(vec->cont[i]);
606         }
607     }
608
609     free(vec->cont);
610     vec->cont = NULL;
611     vec->size = 0;
612
613     return 0;
614 }
```

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().

```

539 {
540     void* ptr=NULL;
541
542     if (vec == NULL)
543     {
544         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
545                 __FILE__, __LINE__);
546         return -1;
547     }
548
549     if (size == -1)
550         size = LSPVEC_INITSIZE;
551
552     if ((ptr = calloc(size, sizeof(DBLabelSwitchedPath*))) == NULL)
553     {
554         addError(CRITICAL, "Critical lack of memory in %s at line %d",
555                 __FILE__, __LINE__);
556         return -1;
557     }
558
559     vec->size = size;
```

```

560     vec->cont = ptr;
561
562     return 0;
563 }

```

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`.

```

510 {
511     DBLSPVec *vec=NULL;
512     void* ptr=NULL;
513
514     if ((vec = calloc(1,sizeof(DBLSPVec))) == NULL)
515     {
516         addError(CRITICAL,"Critical lack of memory in %s at line %d",
517             __FILE__,__LINE__);
518         return NULL;
519     }
520
521     if (size == -1)
522         size = LSPVEC_INITSIZE;
523
524     if ((ptr = calloc(size,sizeof(DBLabelSwitchedPath*))) == NULL)
525     {
526         addError(CRITICAL,"Critical lack of memory in %s at line %d",
527             __FILE__,__LINE__);
528         free(vec);
529         return NULL;
530     }
531
532     vec->size = size;
533     vec->cont = ptr;
534
535     return vec;
536 }

```

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`.

```

690 {
691     if (vec == NULL || vec->cont == NULL ||
692         id < 0 || id >= vec->size || vec->cont[id] == NULL)
693     {
694         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
695             __FILE__,__LINE__);
696         return -1;
697     }
698
699     vec->cont[id]=NULL;
700
701     return 0;
702 }

```

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__);

```

```

664     return -1;
665 }
666
667 if (id >= vec->size)
668 {
669     if (DBlspVecResize(vec,max(2*vec->size,id+1))<0)
670     {
671         addError(CRITICAL,"Unable to resize LSP vector prior to insertion in %s at line %d",
672             __FILE__,__LINE__);
673         return -1;
674     }
675 }
676
677 if (vec->cont[id] != NULL)
678 {
679     addError(CRITICAL,"Trying to add an LSP with a reserved ID in %s at line %d",
680         __FILE__,__LINE__);
681     return -1;
682 }
683
684 vec->cont[id]=lsp;
685
686 return 0;
687 }

```

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().

```

78 {
79     if (node == NULL)
80     {
81         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
82             __FILE__,__LINE__);
83         return -1;
84     }
85
86     longListEnd(&(node->inNeighb));
87     longListEnd(&(node->outNeighb));
88     free(node);
89
90     return 0;
91 }

```

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.

```

50 {
51     if (node == NULL)
52     {
53         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
54             __FILE__,__LINE__);
55         return -1;
56     }
57
58     if (longListInit(&(node->inNeighb),-1) < 0)
59     {
60         addError(CRITICAL,"Unable to initialize the incoming neighbour list in %s at line %d",
61             __FILE__,__LINE__);
62         return -1;
63     }
64
65     if (longListInit(&(node->outNeighb),-1) < 0)
66     {
67         addError(CRITICAL,"Unable to initialize the outgoing neighbour list in %s at line %d",
68             __FILE__,__LINE__);
69         longListEnd(&(node->inNeighb));
70         return -1;
71     }
72
73     return 0;
74 }

```

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().

```

19 {
20     DBNode* ptr=NULL;
21
22     if ((ptr = calloc(1,sizeof(DBNode))) == NULL)
23     {
24         addError(CRITICAL,"Critical lack of memory in %s at line %d",
25             __FILE__,__LINE__);
26         return NULL;
27     }
28
29     if (longListInit(&(ptr->inNeighb),-1) < 0)
30     {
31         addError(CRITICAL,"Unable to initialize the incoming neighbour list in %s at line %d",
32             __FILE__,__LINE__);
33         return NULL;
34     }
35
36     if (longListInit(&(ptr->outNeighb),-1) < 0)
37     {
38         addError(CRITICAL,"Unable to initialize the outgoing neighbour list in %s at line %d",
39             __FILE__,__LINE__);
40         longListEnd(&(ptr->inNeighb));
41         free(ptr);
42         return NULL;
43     }
44
45     return ptr;
46 }

```


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`.

```

350 {
351     int i;
352
353     if (vec == NULL || vec->cont == NULL)
354     {
355         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
356                 __FILE__, __LINE__);
357         return -1;
358     }
359
360     for (i=0; i<vec->size; i++)
361     {
362         if (vec->cont[i]!=NULL)
363         {
364             DBnodeDestroy(vec->cont[i]);
365         }
366     }
367
368     free(vec->cont);
369     free(vec);
370
371     return 0;
372 }
```

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()`.

```

375 {
376     int i;
377
378     if (vec == NULL || vec->cont == NULL)
379     {
380         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
381                 __FILE__, __LINE__);
382         return -1;
383     }
384
385     for (i=0; i<vec->size; i++)
386     {
387         if (vec->cont[i]!=NULL)
388         {
389             DBnodeDestroy(vec->cont[i]);
390         }
391     }
392
393     free(vec->cont);
394     vec->cont = NULL;
395     vec->size = 0;
396     vec->top = 0;
397
398     return 0;
399 }
```

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().

```

322 {
323     void* ptr=NULL;
324
325     if (vec == NULL)
326     {
327         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
328                 __FILE__,__LINE__);
329         return -1;
330     }
331
332     if (size == -1)
333         size = NODEVEC_INITSIZE;
334
335     if ((ptr = calloc(size,sizeof(DBNode*))) == NULL)
336     {
337         addError(CRITICAL,"Critical lack of memory in %s at line %d",
338                 __FILE__,__LINE__);
339         return -1;
340     }
341
342     vec->size = size;
343     vec->top = 0;
344     vec->cont = ptr;
345
346     return 0;
347 }
```

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.

```

291 {
292     DBNodeVec *vec=NULL;
293     void* ptr=NULL;
294
295     if ((vec = calloc(1,sizeof(DBNodeVec))) == NULL)
296     {
297         addError(CRITICAL,"Critical lack of memory in %s at line %d",
298                 __FILE__,__LINE__);
299         return NULL;
300     }
301
302     if (size == -1)
303         size = NODEVEC_INITSIZE;
304
305     if ((ptr = calloc(size,sizeof(DBNode*))) == NULL)
306     {
307         addError(CRITICAL,"Critical lack of memory in %s at line %d",
308                 __FILE__,__LINE__);
309         free(vec);
310         return NULL;
311     }
```

```

311     }
312
313     vec->size = size;
314     vec->top = 0;
315     vec->cont = ptr;
316
317     return vec;
318 }

```

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()`.

```

486 {
487     if (vec == NULL || vec->cont == NULL ||
488         id < 0 || id >= vec->size || vec->cont[id] == NULL)
489     {
490         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
491             __FILE__, __LINE__);
492         return -1;
493     }
494
495     DBnodeDestroy(vec->cont[id]);
496     vec->cont[id]=NULL;
497
498     while (vec->cont[vec->top-1] == NULL)
499         vec->top--;
500
501     return 0;
502 }

```

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()`.

```

402 {
403     void *ptr=NULL;
404     int i;
405
406     if (vec == NULL || vec->cont == NULL)
407     {
408         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
409             __FILE__, __LINE__);
410         return -1;
411     }
412
413     if (size < vec->size)
414     {
415         for (i=size;i<vec->size;i++)
416         {
417             if (vec->cont[i]!=NULL)

```

```

418         {
419             DBnodeDestroy(vec->cont[i]);
420             vec->cont[i]=NULL;
421         }
422     }
423
424     if (size < vec->top)
425     {
426         vec->top = size;
427         while (vec->cont[vec->top-1] == NULL)
428             vec->top--;
429     }
430
431 }
432
433
434
435 if ((ptr = realloc(vec->cont, size * sizeof(DBNode*))) == NULL)
436 {
437     addError(CRITICAL,"Critical lack of memory in %s at line %d",
438             __FILE__,__LINE__);
439     return -1;
440 }
441
442 if (size > vec->size)
443 {
444     memset(ptr + (vec->size * sizeof(DBNode*)), 0, (size-vec->size) * sizeof(DBNode*));
445 }
446
447 vec->size=size;
448 vec->cont=ptr;
449
450 return 0;
451 }

```

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().

```

454 {
455     if (vec == NULL || vec->cont == NULL || node == NULL || id < 0)
456     {
457         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
458                 __FILE__,__LINE__);
459         return -1;
460     }
461
462     if (id >= vec->size)
463     {
464         if (DBnodeVecResize(vec,max(2*vec->size,id+1))<0)
465         {
466             addError(CRITICAL,"Unable to resize node vector prior to insertion in %s at line %d",
467                     __FILE__,__LINE__);
468             return -1;
469         }
470     }
471
472     if (vec->cont[id] != NULL)
473     {

```

```

474         addError(CRITICAL,"Trying to add a node with a reserved ID in %s at line %d",
475                 __FILE__,__LINE__);
476         return -1;
477     }
478
479     vec->cont[id]=node;
480     vec->top = max(vec->top, id+1);
481
482     return 0;
483 }

```

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()`.

```

238 {
239     long i,oa;
240     double ptot,rtot;
241     DBLabelSwitchedPath* lsp=NULL;
242
243     if (link == NULL)
244     {
245         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
246                 __FILE__,__LINE__);
247         return;
248     }
249
250     printf("\tList of LSPs\n");
251     printf("\t-----\n");
252
253     for (i=0; i<link->lspList.top; ++i)
254     {
255         lsp = link->lspList.cont[i];
256         printf("%ld ", lsp->id);
257     }
258
259     printf("\n\n");
260
261     printf("\tLink-state\n");
262     printf("\t-----\n");
263
264     for (oa=0; oa<NB_OA; ++oa)
265     {
266         ptot = 0;
267         rtot = 0;
268
269         printf("\tCapacity[%ld] = %f\n", oa, link->state.cap[oa]);
270
271         for (i=0; i<NB_PREEMPTION; ++i)
272         {
273             ptot += link->state.pbw[oa][i];
274             rtot += link->state.rbw[oa][i];
275         }
276
277         printf("\ttpbw[%ld] = %f\n", oa, ptot);
278         printf("\trbw[%ld] = %f\n", oa, rtot);
279     }
280

```

```
281     printf("\n\n");
282
283 }
```

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.

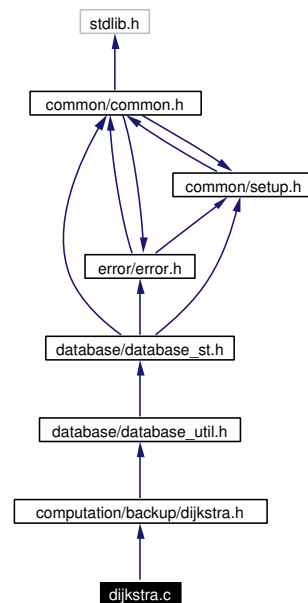
Referenced by DBprintDB().

```
111 {
112     long i;
113
114     if (node == NULL)
115     {
116         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
117                 __FILE__, __LINE__);
118         return;
119     }
120
121     printf("Incoming neighbors : \n");
122
123     for (i=0; i<node->inNeighb.top; i++)
124     {
125         printf("%ld ", node->inNeighb.cont[i]);
126     }
127
128     printf("\nOutgoing neighbors : \n");
129
130     for (i=0; i<node->outNeighb.top; i++)
131     {
132         printf("%ld ", node->outNeighb.cont[i]);
133     }
134
135     printf("\n");
136 }
```

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 * CPopTop (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`.

```

88 {
89     if (pq == NULL)
90     {
91         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
92                 __FILE__, __LINE__);
93         return -1;
94     }

```

```

95
96     if (CPendPQ(pq) < -1)
97     {
98         addError(CRITICAL,"Destruction incomplete in %s at line %d",
99             __FILE__,__LINE__);
100         return -1;
101     }
102
103     free(pq);
104
105     return 0;
106
107 }

```

4.17.1.2 int CPdestroyTN (CPTreeNode * tn)

Definition at line 17 of file dijkstra.c.

References addError(), CRITICAL, and free.

Referenced by CPpopTop().

```

18 {
19     if (tn == NULL)
20     {
21         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
22             __FILE__,__LINE__);
23         return -1;
24     }
25
26     free(tn);
27
28     return 0;
29 }

```

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().

```

69 {
70     if (pq == NULL)
71     {
72         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
73             __FILE__,__LINE__);
74         return -1;
75     }
76
77     while ((CPpopTop(pq)) != NULL);
78
79     pq->size = 0;
80     pq->root = NULL;
81     pq->top = NULL;
82
83     return 0;
84
85 }

```


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().

```

33 {
34     if (pq == NULL)
35     {
36         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
37                 __FILE__, __LINE__);
38         return -1;
39     }
40
41     pq->root = NULL;
42     pq->top = NULL;
43     pq->size = 0;
44
45     return 0;
46 }
```

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().

```

110 {
111     CPTreeNode* tn, *ptr, *lastPtr=NULL;
112
113     if (pq == NULL)
114     {
115         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
116                 __FILE__, __LINE__);
117         return -1;
118     }
119
120     if ((tn = CPnewTN()) == NULL)
121     {
122         addError(CRITICAL, "Impossible to allocated a new TreeNode in %s at line %d",
123                 __FILE__, __LINE__);
124         return -1;
125     }
126
127     tn->key = key;
128     tn->node = dn;
129
130     // size == 0
131     if (pq->root == NULL)
132     {
133         pq->root = tn;
134         pq->top = tn;
135         pq->size++;
136         return 0;
137     }
138
139     // lower than top
```

```

140     if (key <= pq->top->key)
141     {
142         pq->top->leq = tn;
143         tn->father = pq->top;
144         pq->top = tn;
145         pq->size++;
146         return 0;
147     }
148
149     // anywhere else ...
150     ptr = pq->root;
151
152     while (ptr != NULL)
153     {
154         lastPtr = ptr;
155
156         if (key <= ptr->key)
157             ptr = ptr->leq;
158         else
159             ptr = ptr->gt;
160     }
161
162     if (key <= lastPtr->key)
163         lastPtr->leq = tn;
164     else
165         lastPtr->gt = tn;
166
167     tn->father = lastPtr;
168     pq->size++;
169
170     return 0;
171 }

```

4.17.1.6 **CPPrioQueue*** CPnewPQ ()

Definition at line 48 of file dijkstra.c.

References `addError()`, `calloc`, and `CRITICAL`.

```

49 {
50     CPPrioQueue* pq=NULL;
51
52     if ((pq = calloc(1, sizeof(CPPrioQueue))) == NULL)
53     {
54         addError(CRITICAL,"Impossible to allocated a new PrioQueue in %s at line %d",
55             __FILE__,__LINE__);
56         return NULL;
57     }
58
59     /* Done by calloc !!!
60     pq->root = NULL;
61     pq->top = NULL;
62     pq->size = 0;
63     */
64
65     return pq;
66 }

```

4.17.1.7 **CPTreeNode*** CPnewTN ()

Definition at line 3 of file dijkstra.c.

References addError(), calloc, and CRITICAL.

Referenced by CPinsertPQ().

```

4 {
5     CPTreeNode* tn=NULL;
6
7     if ((tn = calloc(1, sizeof(CPTreeNode))) == NULL)
8     {
9         addError(CRITICAL, "Impossible to allocated a new PrioQueue in %s at line %d",
10                __FILE__, __LINE__);
11         return NULL;
12     }
13
14     return tn;
15 }
```

4.17.1.8 CPDijkNode* CPopTop (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().

```

174 {
175     CPTreeNode* tn;
176     CPDijkNode* dn;
177
178     if (pq == NULL)
179     {
180         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
181                __FILE__, __LINE__);
182         return NULL;
183     }
184
185     tn = pq->top;
186
187     if (tn != NULL)
188     {
189         pq->size--;
190
191         if (tn == pq->root)
192         {
193             pq->root = tn->gt;
194             pq->top = tn->gt;
195             if (tn->gt; != NULL)
196                 tn->gt;->father = NULL;
197         }
198         else
199         {
200             tn->father->leq = tn->gt;
201             if (tn->gt; != NULL)
202             {
203                 pq->top = tn->gt;
204                 tn->gt;->father = tn->father;
205             }
206             else
207                 pq->top = tn->father;
208         }
209
210         // now find the new top;

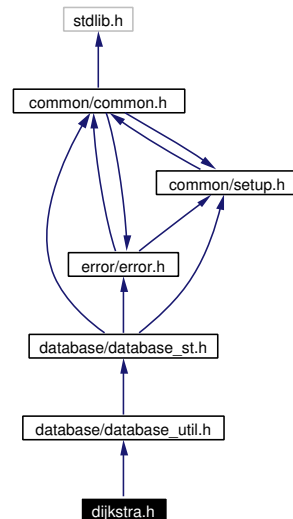
```

```
211         if (pq->size > 0)
212             while (pq->top->leq != NULL)
213                 pq->top = pq->top->leq;
214     }
215     else
216     {
217         return NULL;
218     }
219
220     dn = tn->node;
221
222     if (CPdestroyTN(tn) < 0)
223     {
224         addError(WARNING, "Unable to destroy TreeNode but DijkNode was returned in %s at line %d",
225                 __FILE__, __LINE__);
226     }
227
228     return dn;
229 }
```

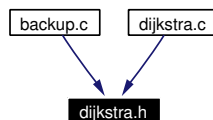
4.18 dijkstra.h File Reference

```
#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.

```

88 {
89     if (pq == NULL)
90     {
91         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
92                 __FILE__, __LINE__);
93         return -1;
94     }
95
96     if (CPendPQ(pq) < -1)
97     {
98         addError(CRITICAL, "Destruction incomplete in %s at line %d",
99                 __FILE__, __LINE__);
100        return -1;
101    }
102
103    free(pq);
104
105    return 0;
106
107 }
```

4.18.2.2 int CPdestroyTN (CPTreeNode *)

Definition at line 17 of file dijkstra.c.

References addError(), CRITICAL, and free.

Referenced by CPpopTop().

```

18 {
19     if (tn == NULL)
20     {
21         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
22                 __FILE__, __LINE__);
23         return -1;
24     }
25 }
```

```

26     free(tn);
27
28     return 0;
29 }

```

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()`.

```

69 {
70     if (pq == NULL)
71     {
72         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
73                 __FILE__, __LINE__);
74         return -1;
75     }
76
77     while ((CPpopTop(pq)) != NULL);
78
79     pq->size = 0;
80     pq->root = NULL;
81     pq->top = NULL;
82
83     return 0;
84
85 }

```

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()`.

```

33 {
34     if (pq == NULL)
35     {
36         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
37                 __FILE__, __LINE__);
38         return -1;
39     }
40
41     pq->root = NULL;
42     pq->top = NULL;
43     pq->size = 0;
44
45     return 0;
46 }

```

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().

```

110 {
111     CPTreeNode* tn, *ptr, *lastPtr=NULL;
112
113     if (pq == NULL)
114     {
115         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
116             __FILE__, __LINE__);
117         return -1;
118     }
119
120     if ((tn = CPnewTN()) == NULL)
121     {
122         addError(CRITICAL,"Impossible to allocated a new TreeNode in %s at line %d",
123             __FILE__, __LINE__);
124         return -1;
125     }
126
127     tn->key = key;
128     tn->node = dn;
129
130     // size == 0
131     if (pq->root == NULL)
132     {
133         pq->root = tn;
134         pq->top = tn;
135         pq->size++;
136         return 0;
137     }
138
139     // lower than top
140     if (key <= pq->top->key)
141     {
142         pq->top->leq = tn;
143         tn->father = pq->top;
144         pq->top = tn;
145         pq->size++;
146         return 0;
147     }
148
149     // anywhere else ...
150     ptr = pq->root;
151
152     while (ptr != NULL)
153     {
154         lastPtr = ptr;
155
156         if (key <= ptr->key)
157             ptr = ptr->leq;
158         else
159             ptr = ptr->gt;
160     }
161
162     if (key <= lastPtr->key)
163         lastPtr->leq = tn;
164     else
165         lastPtr->gt = tn;
166
167     tn->father = lastPtr;
168     pq->size++;
169
170     return 0;

```



```
171 }
```

4.18.2.6 [CPPrioQueue*](#) CPnewPQ ()

Definition at line 48 of file dijkstra.c.

References [addError\(\)](#), [calloc](#), and [CRITICAL](#).

```
49 {
50     CPPrioQueue* pq=NULL;
51
52     if ((pq = calloc(1, sizeof(CPPrioQueue))) == NULL)
53     {
54         addError(CRITICAL,"Impossible to allocated a new PrioQueue in %s at line %d",
55                 __FILE__,__LINE__);
56         return NULL;
57     }
58
59     /* Done by calloc !!!
60     pq->root = NULL;
61     pq->top = NULL;
62     pq->size = 0;
63     */
64
65     return pq;
66 }
```

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\(\)](#).

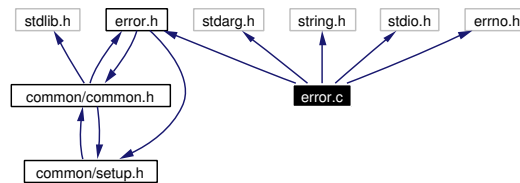
```
174 {
175     CPTreeNode* tn;
176     CPDijkNode* dn;
177
178     if (pq == NULL)
179     {
180         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
181                 __FILE__,__LINE__);
182         return NULL;
183     }
184
185     tn = pq->top;
186
187     if (tn != NULL)
188     {
189         pq->size--;
190
191         if (tn == pq->root)
192         {
193             pq->root = tn->gt;
194             pq->top = tn->gt;
195             if (tn->gt; != NULL)
196                 tn->gt;->father = NULL;
197         }
198     }
```

```
198     else
199     {
200         tn->father->leq = tn->gt;
201         if (tn->gt != NULL)
202         {
203             pq->top = tn->gt;
204             tn->gt->father = tn->father;
205         }
206         else
207             pq->top = tn->father;
208     }
209
210     // now find the new top;
211     if (pq->size > 0)
212         while (pq->top->leq != NULL)
213             pq->top = pq->top->leq;
214 }
215 else
216 {
217     return NULL;
218 }
219
220 dn = tn->node;
221
222 if (CPdestroyTN(tn) < 0)
223 {
224     addError(WARNING, "Unable to destroy TreeNode but DijkNode was returned in %s at line %d",
225             __FILE__, __LINE__);
226 }
227
228 return dn;
229 }
```

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().

```

41 {
42     va_list lst;
43     void *ptr=NULL;
44     char tmpmsg[ERRORMSG_SIZE];
45
46     va_start(lst,msg);
47
48     vsnprintf(tmpmsg,ERRORMSG_SIZE,msg,lst);
49     tmpmsg[ERRORMSG_SIZE-1]='\0';
50
51
52     switch (level)
53     {
54         case INFO:
55         case WARNING:
56         case CRITICAL:
57         case PANIC:
58             break;
59     }
60
61     if (errorManager.top >= errorManager.size-ERROR_PROVISION)
62     {
63         if (( ptr = realloc(errorManager.list, errorManager.size *
64                             2 * sizeof(ErrorElem))) == NULL)
65         {
66             if (errorManager.top < errorManager.size)
67             {
68                 errorManager.list[errorManager.top].gravity = CRITICAL;
69                 strncpy(errorManager.list[errorManager.top].message,

```

```

70             "Critical lack of memory encountered while resizing error manager",
71             ERRORMSG_SIZE);
72         errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
73     }
74     else
75         addError(PANIC,"");
76
77     if (errorManager.top < errorManager.size)
78     {
79         errorManager.list[errorManager.top].gravity = level;
80         strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
81         errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
82     }
83 }
84 else
85 {
86     errorManager.list=ptr;
87     errorManager.size*=2;
88     errorManager.list[errorManager.top].gravity = level;
89     strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
90     errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
91 }
92 }
93 else
94 {
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 }

```

4.19.2.2 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 }

```

4.19.2.3 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 }

```

4.19.2.4 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.

```
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] ");
117                 break;
118             case CRITICAL:
119                 printf("[CRITICAL] ");
120                 break;
121             case PANIC:
122                 printf("[PANIC] ");
123
124         }
125
126         printf("%s\n", error.message);
127     }
128 }
129 }
```

4.19.3 Variable Documentation

4.19.3.1 [ErrorList errorManager](#)

Definition at line 26 of file error.c.

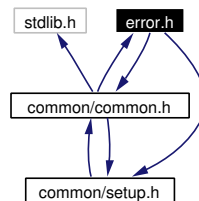
Referenced by addError(), errorDestroy(), errorInit(), and printErrorStack().

4.20 error.h File Reference

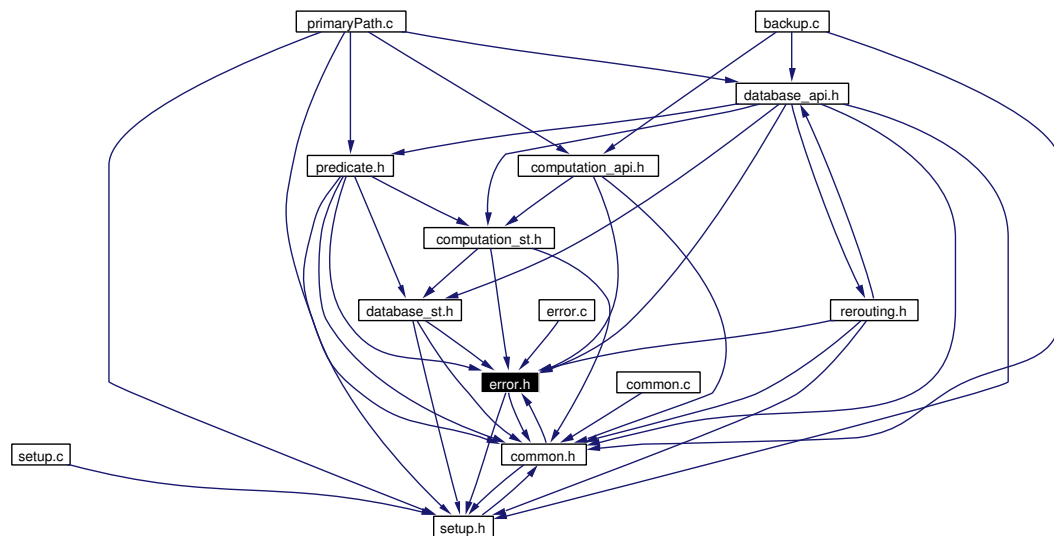
```
#include "common/common.h"
```

```
#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().

```

41 {
42     va_list lst;
43     void *ptr=NULL;
44     char tmpmsg[ERRORMSG_SIZE];
45
46     va_start(lst,msg);
47
48     vsnprintf(tmpmsg,ERRORMSG_SIZE,msg,lst);
49     tmpmsg[ERRORMSG_SIZE-1]='\0';
50
51
52     switch (level)
53     {
54         case INFO:
55         case WARNING:
56         case CRITICAL:
57         case PANIC:
58             break;
59     }
60
61     if (errorManager.top >= errorManager.size-ERROR_PROVISION)
62     {
63         if (( ptr = realloc(errorManager.list, errorManager.size *
64             2 * sizeof(ErrorElem))) == NULL)
65         {
66             if (errorManager.top < errorManager.size)
67             {
68                 errorManager.list[errorManager.top].gravity = CRITICAL;
69                 strncpy(errorManager.list[errorManager.top].message,
70                     "Critical lack of memory encountered while resizing error manager",
71                     ERRORMSG_SIZE);
72                 errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
73             }
74             else
75                 addError(PANIC,"");
76
77             if (errorManager.top < errorManager.size)
78             {
79                 errorManager.list[errorManager.top].gravity = level;
80                 strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
81                 errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
82             }
83         }
84         else
85         {
86             errorManager.list=ptr;
87             errorManager.size*=2;
88             errorManager.list[errorManager.top].gravity = level;
89             strncpy(errorManager.list[errorManager.top].message,tmpmsg,ERRORMSG_SIZE);
90             errorManager.list[errorManager.top++].message[ERRORMSG_SIZE-1]='\0';
91         }
92     }
93     else
94     {

```

```

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] ");

```

```
117         break;
118     case CRITICAL:
119         printf("[CRITICAL] ");
120         break;
121     case PANIC:
122         printf("[PANIC] ");
123
124     }
125
126     printf("%s\n", error.message);
127
128 }
129 }
```

4.20.4 Variable Documentation

4.20.4.1 [ErrorList errorManager](#)

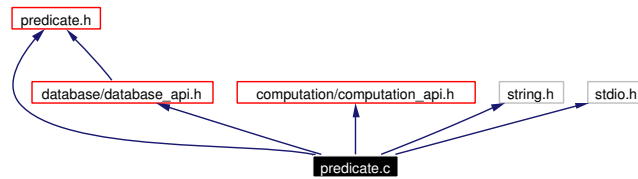
Definition at line 19 of file error.h.

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, DBLabelSwitched-Path *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    }
```

```

23
24     for (i=0;i<NB_OA;i++)
25     {
26         if (ls->cap[i]<occupied[i])
27             return FALSE;
28         if (ls->cap[i]<total[i])
29             gain[i]=total[i]-ls->cap[i];
30         else
31             gain[i]=0;
32     }
33
34     return TRUE;
35 }

```

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().

```

38 {
39     int i;
40
41     for (i=0;i<req->forbidLinks.top;i++)
42     {
43         if (req->forbidLinks.cont[i]==ls->color)
44             return FALSE;
45     }
46
47     return TRUE;
48 }

```

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().

```

170 {
171     LSPRequest req;
172     bool gate;
173
174     if (ls == NULL || lsp == NULL || gain==NULL)
175     {
176         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
177             __FILE__, __LINE__);
178         return FALSE;
179     }
180 }

```

```

179     }
180
181     if (lspRequestInit(&req)<0)
182     {
183         addError(CRITICAL,"Unable to initialize request in %s at line %d",
184             __FILE__,__LINE__);
185         return FALSE;
186     }
187
188     req.id=lsp->id;
189     req.precedence=lsp->precedence;
190     req.type=lsp->type;
191     req.primID=lsp->primID;
192     req.rerouteInfo.id=lsp->noContentionId;
193     req.rerouteInfo.src=-1;
194     req.rerouteInfo.dst=-1;
195     memcpy(req.bw,lsp->bw, NB_OA * sizeof(double));
196
197     if (longListCopy(&(req.forbidLinks),&(lsp->forbidLinks))<0)
198     {
199         addError(CRITICAL,"Unable to initialize request in %s at line %d",
200             __FILE__,__LINE__);
201         lspRequestEnd(&req);
202         return FALSE;
203     }
204
205     if (longListCopy(&(req.path),&(lsp->path))<0)
206     {
207         addError(CRITICAL,"Unable to initialize request in %s at line %d",
208             __FILE__,__LINE__);
209         lspRequestEnd(&req);
210         return FALSE;
211     }
212
213
214     gate=isValidRequestLink(dataBase,src,dst,ls,&req,gain);
215
216     lspRequestEnd(&req);
217
218     return gate;
219 }

```

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().

```

51 {
52     if (req->id == 1781) {
53         int olivier;
54         fprintf(stderr,"\nbackup %ld: ", req->id);
55         for (olivier=0;olivier<req->path.top;olivier++){
56             fprintf(stderr,"%ld ", req->path.cont[olivier]);
57         }
58     }

```

```

59
60     LongList recPath;
61     DBLinkState newLS;
62     bool gate=TRUE,tmpgate,completePath=TRUE;
63     long i;
64
65     if (ls == NULL || req == NULL || gain==NULL)
66     {
67         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
68             __FILE__,__LINE__);
69         return FALSE;
70     }
71
72     if (DBlinkStateInit(&newLS)<0)
73     {
74         addError(CRITICAL,"Unable to initialize link state in %s at line %d",
75             __FILE__,__LINE__);
76         return FALSE;
77     }
78
79     for (i=0;i<req->path.top && completePath;i++)
80     {
81         if (req->path.cont[i]<0)
82         {
83             completePath=FALSE;
84         }
85     }
86
87     if (! completePath)
88     {
89         if (longListInit(&recPath,-1)<0)
90         {
91             addError(CRITICAL,"Unable to initialize path record structure in %s at line %d",
92                 __FILE__,__LINE__);
93             return FALSE;
94         }
95         if (longListCopy(&recPath,&(req->path))<0)
96         {
97             addError(CRITICAL,"Unable to copy path into record structure in %s at line %d",
98                 __FILE__,__LINE__);
99             longListEnd(&recPath);
100             return FALSE;
101         }
102         req->path.top=0;
103         if (longListPushBack(&(req->path),src)<0 ||
104             longListPushBack(&(req->path),dst)<0)
105         {
106             addError(CRITICAL,"Unable to forge path into request in %s at line %d",
107                 __FILE__,__LINE__);
108             longListEnd(&recPath);
109             return FALSE;
110         }
111     }
112
113     if (req->id == 1781) {
114         int olivier;
115         fprintf(stderr,"\nbackup %ld: ", req->id);
116         for (olivier=0;olivier<req->path.top;olivier++){
117             fprintf(stderr,"%ld ", req->path.cont[olivier]);
118         }
119     }
120
121
122     if (DBevalLSONSetup(dataBase,src,dst,&newLS,ls,req)<0)
123     {
124         addError(CRITICAL,"Unable to update link state in %s at line %d",
125             __FILE__,__LINE__);

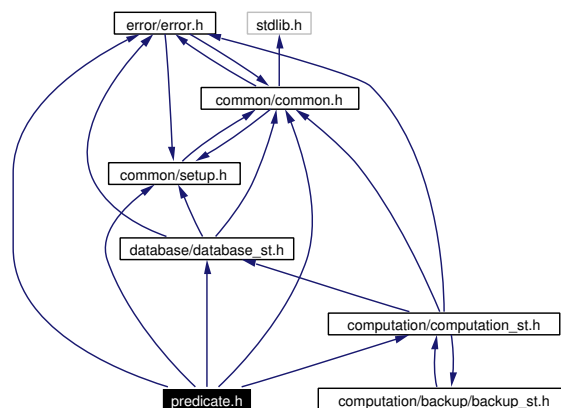
```

```
126         if (!completePath)
127         {
128             longListEnd(&recPath);
129         }
130         return FALSE;
131     }
132
133     if (damoteConfig.predicateConfig.capacityClause)
134     {
135         tmpgate=capacityClause(&newLS,req,gain);
136         gate=gate && tmpgate;
137     }
138
139     if (damoteConfig.predicateConfig.colorClause)
140     {
141         tmpgate=colorClause(&newLS,req);
142         gate=gate && tmpgate;
143     }
144
145     if (req->rerouteInfo.id >= 0 && req->rerouteInfo.src==src
146         && req->rerouteInfo.dst==dst)
147     {
148         gate= FALSE;
149     }
150
151     if (!completePath)
152     {
153         if (longListCopy(&(req->path),&recPath)<0)
154         {
155             addError(CRITICAL,"Unable to restore path into request in %s at line %d",
156                     __FILE__,__LINE__);
157             longListEnd(&recPath);
158             return FALSE;
159         }
160         longListEnd(&recPath);
161     }
162
163     DBlinkStateEnd(&newLS);
164
165     return gate;
166 }
```

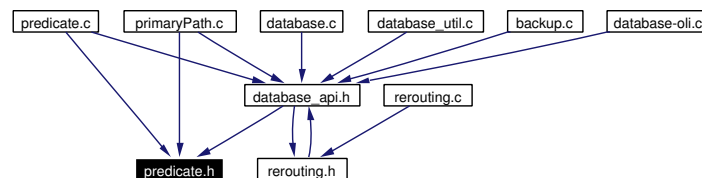

4.22 predicate.h File Reference

```
#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`, `LSPPrerouteInfo::dst`, `FALSE`, `DBLabelSwitchedPath::forbidLinks`, `LSPRequest::forbidLinks`, `LSPPrerouteInfo::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().

```

170 {
171     LSPRequest req;
172     bool gate;
173
174     if (ls == NULL || lsp == NULL || gain==NULL)
175     {
176         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
177             __FILE__, __LINE__);
178         return FALSE;
179     }
180
181     if (lspRequestInit(&req)<0)
182     {
183         addError(CRITICAL,"Unable to initialize request in %s at line %d",
184             __FILE__, __LINE__);
185         return FALSE;
186     }
187
188     req.id=lsp->id;
189     req.precedence=lsp->precedence;
190     req.type=lsp->type;
191     req.primID=lsp->primID;
192     req.rerouteInfo.id=lsp->noContentionId;
193     req.rerouteInfo.src=-1;
194     req.rerouteInfo.dst=-1;
195     memcpy(req.bw,lsp->bw, NB_OA * sizeof(double));
196
197     if (longListCopy(&(req.forbidLinks),&(lsp->forbidLinks))<0)
198     {
199         addError(CRITICAL,"Unable to initialize request in %s at line %d",
200             __FILE__, __LINE__);
201         lspRequestEnd(&req);
202         return FALSE;
203     }
204
205     if (longListCopy(&(req.path),&(lsp->path))<0)
206     {
207         addError(CRITICAL,"Unable to initialize request in %s at line %d",
208             __FILE__, __LINE__);
209         lspRequestEnd(&req);
210         return FALSE;
211     }
212
213
214     gate=isValidRequestLink(dataBase,src,dst,ls,&req,gain);
215
216     lspRequestEnd(&req);
217
218     return gate;
219 }
```

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().

```

51 {
52     if (req->id == 1781) {
53         int olivier;
54         fprintf(stderr, "\nbackup %ld: ", req->id);
55         for (olivier=0; olivier<req->path.top; olivier++){
56             fprintf(stderr, "%ld ", req->path.cont[olivier]);
57         }
58     }
59
60     LongList recPath;
61     DBlinkState newLS;
62     bool gate=TRUE, tmpgate, completePath=TRUE;
63     long i;
64
65     if (ls == NULL || req == NULL || gain==NULL)
66     {
67         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
68                 __FILE__, __LINE__);
69         return FALSE;
70     }
71
72     if (DBlinkStateInit(&newLS)<0)
73     {
74         addError(CRITICAL, "Unable to initialize link state in %s at line %d",
75                 __FILE__, __LINE__);
76         return FALSE;
77     }
78
79     for (i=0; i<req->path.top && completePath; i++)
80     {
81         if (req->path.cont[i]<0)
82         {
83             completePath=FALSE;
84         }
85     }
86
87     if (! completePath)
88     {
89         if (longListInit(&recPath, -1)<0)
90         {
91             addError(CRITICAL, "Unable to initialize path record structure in %s at line %d",
92                     __FILE__, __LINE__);
93             return FALSE;
94         }
95         if (longListCopy(&recPath, &(req->path))<0)
96         {
97             addError(CRITICAL, "Unable to copy path into record structure in %s at line %d",
98                     __FILE__, __LINE__);
99             longListEnd(&recPath);
100             return FALSE;
101         }
102         req->path.top=0;
103         if (longListPushBack(&(req->path), src)<0 ||
104             longListPushBack(&(req->path), dst)<0)
105         {
106             addError(CRITICAL, "Unable to forge path into request in %s at line %d",
107                     __FILE__, __LINE__);
108             longListEnd(&recPath);

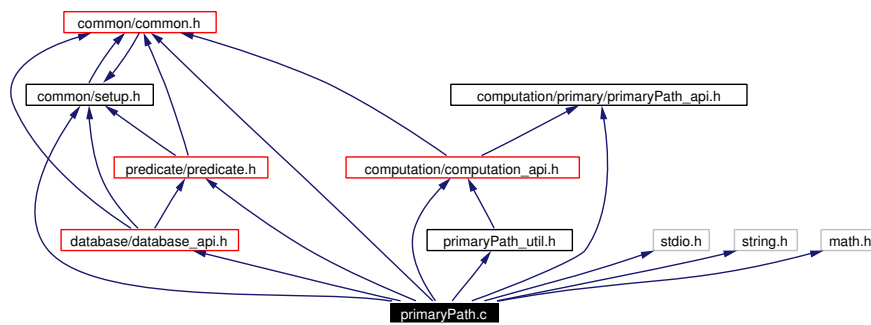
```

```
109         return FALSE;
110     }
111 }
112
113 if (req->id == 1781) {
114     int olivier;
115     fprintf(stderr, "\nbackup %ld: ", req->id);
116     for (olivier=0; olivier<req->path.top; olivier++){
117         fprintf(stderr, "%ld ", req->path.cont[olivier]);
118     }
119 }
120
121
122 if (DBevalLSONSetup(dataBase, src, dst, &newLS, ls, req) < 0)
123 {
124     addError(CRITICAL, "Unable to update link state in %s at line %d",
125             __FILE__, __LINE__);
126     if (!completePath)
127     {
128         longListEnd(&recPath);
129     }
130     return FALSE;
131 }
132
133 if (damoteConfig.predicateConfig.capacityClause)
134 {
135     tmpgate=capacityClause(&newLS, req, gain);
136     gate=gate && tmpgate;
137 }
138
139 if (damoteConfig.predicateConfig.colorClause)
140 {
141     tmpgate=colorClause(&newLS, req);
142     gate=gate && tmpgate;
143 }
144
145 if (req->rerouteInfo.id >= 0 && req->rerouteInfo.src==src
146     && req->rerouteInfo.dst==dst)
147 {
148     gate= FALSE;
149 }
150
151 if (!completePath)
152 {
153     if (longListCopy(&(req->path), &recPath) < 0)
154     {
155         addError(CRITICAL, "Unable to restore path into request in %s at line %d",
156                 __FILE__, __LINE__);
157         longListEnd(&recPath);
158         return FALSE;
159     }
160     longListEnd(&recPath);
161 }
162
163 DBlinkStateEnd(&newLS);
164
165 return gate;
166 }
```

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 `DAMOTE-Config::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().

```

1172 {
1173     LongList activeNodes;
1174     BKNode *tmpNode;
1175     bool done=FALSE;
1176     int *activeFlags;
1177     long src,i,j,k,nd,top,threshold,size,iter=0;
1178     double tmpCost;
1179
1180
1181     if (topo == NULL)
1182     {
1183         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1184             __FILE__,__LINE__);
1185         return -1;
1186     }
1187
1188     if ((src=getRequestSrc(req))<0)
1189     {
1190         addError(CRITICAL,"Unable to get requested source in %s at line %d",
1191             __FILE__,__LINE__);
1192         return -1;
1193     }
1194
1195     size=topo->nodeInd.top;
1196
1197     if (longListInit(&activeNodes,size)<0)
1198     {
1199         addError(CRITICAL,"Unable to initialize the active nodes list in %s at line %d",
1200             __FILE__,__LINE__);
1201         return -1;
1202     }
1203
1204     if ((activeFlags = (int*) calloc(size,sizeof(long))) == NULL)
1205     {
1206         addError(CRITICAL,"Critical lack of memory in %s at line %d",
1207             __FILE__,__LINE__);
1208         longListEnd(&activeNodes);
1209         return -1;
1210     }
1211
1212     if (src>=size)
1213     {
1214         addError(CRITICAL,"Inexistent node in %s at line %d",
1215             __FILE__,__LINE__);
1216         longListEnd(&activeNodes);
1217         free(activeFlags);
1218         return -1;
1219     }
1220     if (initScore(src,topo)<0)
1221     {
1222         addError(CRITICAL,"Unable to initialize scores in %s at line %d",
1223             __FILE__,__LINE__);
1224         longListEnd(&activeNodes);

```

```

1225         free(activeFlags);
1226         return -1;
1227     }
1228     topo=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.top;
1229     for (i=0;i<top;i++)
1230     {
1231         nd=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.cont[i].neighbId;
1232         if (nd>=size)
1233         {
1234             addError(CRITICAL,"Inexistent node in %s at line %d",
1235                     __FILE__,__LINE__);
1236             longListEnd(&activeNodes);
1237             free(activeFlags);
1238             return -1;
1239         }
1240
1241         if (longListPushBack(&activeNodes,nd)<0)
1242         {
1243             addError(CRITICAL,"Undetermined error in %s at line %d",
1244                     __FILE__,__LINE__);
1245             longListEnd(&activeNodes);
1246             free(activeFlags);
1247             return -1;
1248         }
1249
1250         if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
1251         {
1252             addError(CRITICAL,"Undetermined error in %s at line %d",
1253                     __FILE__,__LINE__);
1254             longListEnd(&activeNodes);
1255             free(activeFlags);
1256             return -1;
1257         }
1258         for (k=0;(k<tmpNode->inNeighb.top) && (tmpNode->inNeighb.cont[k].neighbId!=src);k++);
1259         if (k>=tmpNode->inNeighb.top)
1260         {
1261             addError(CRITICAL,"Topology unconsistency in %s at line %d",
1262                     __FILE__,__LINE__);
1263             longListEnd(&activeNodes);
1264             free(activeFlags);
1265             return -1;
1266         }
1267         tmpNode->info.cost=makeScore(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
1268         tmpNode->info.newCost=tmpNode->info.cost;
1269         tmpNode->neighbInd=k;
1270         tmpNode->info.newNeighbInd=tmpNode->neighbInd;
1271         updateNodeInfoOnElect(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
1272         activateNodeInfo(topo,nd);
1273         activeFlags[nd]=1;
1274     }
1275     activeFlags[src]=2;
1276
1277     while (!done)
1278     {
1279         iter++;
1280         done=TRUE;
1281         threshold=activeNodes.top;
1282         for (i=0;i<threshold;i++)
1283         {
1284             topo=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.top;
1285             for (j=0;j<top;j++)
1286             {
1287                 nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.cont[j].neighbId;
1288                 if (nd>=size)
1289                 {
1290                     addError(CRITICAL,"Inexistent node in %s at line %d",
1291                             __FILE__,__LINE__);

```



```

1292         longListEnd(&activeNodes);
1293         free(activeFlags);
1294         return -1;
1295     }
1296
1297     if (activeFlags[nd]!=0 && noLoop(topo,nd,activeNodes.cont[i]))
1298     {
1299         tmpCost=makeScore(topo,req,nd,activeNodes.cont[i],
1300                           &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost=tmpC
1301         if (tmpCost-topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost
1302         {
1303             done=FALSE;
1304             topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost=tmpC
1305             topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd
1306             updateNodeInfoOnElect(topo,req,nd,activeNodes.cont[i],
1307                                   &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont
1308         }
1309     }
1310 }
1311
1312
1313 if (activeFlags[activeNodes.cont[i]]==1)
1314 {
1315     top=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.top;
1316     for (j=0;j<top;j++)
1317     {
1318         nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.cont[j].
1319         if (nd>=size)
1320         {
1321             addError(CRITICAL,"Inexistent node in %s at line %d",
1322                     __FILE__,__LINE__);
1323             longListEnd(&activeNodes);
1324             free(activeFlags);
1325             return -1;
1326         }
1327
1328         if (activeFlags[nd]==0)
1329         {
1330             done=FALSE;
1331
1332             if (longListPushBack(&activeNodes,nd)<0)
1333             {
1334                 addError(CRITICAL,"Undetermined error in %s at line %d",
1335                         __FILE__,__LINE__);
1336                 longListEnd(&activeNodes);
1337                 free(activeFlags);
1338                 return -1;
1339             }
1340
1341             if((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
1342             {
1343                 addError(CRITICAL,"Undetermined error in %s at line %d",
1344                         __FILE__,__LINE__);
1345                 longListEnd(&activeNodes);
1346                 free(activeFlags);
1347                 return -1;
1348             }
1349             for (k=0;(k<tmpNode->inNeighb.top) &&
1350                  (tmpNode->inNeighb.cont[k].neighbId!=activeNodes.cont[i];k++);
1351             if (k>=tmpNode->inNeighb.top)
1352             {
1353                 addError(CRITICAL,"Topology unconsistency in %s at line %d",
1354                         __FILE__,__LINE__);
1355                 longListEnd(&activeNodes);
1356                 free(activeFlags);
1357                 return -1;
1358             }

```

```

1359         tmpNode->info.cost=makeScore(topo,req,activeNodes.cont[i],nd,&tmpNode->inNeig
1360         tmpNode->info.newCost=tmpNode->info.cost;
1361         tmpNode->neighbInd=k;
1362         tmpNode->info.newNeighbInd=tmpNode->neighbInd;
1363         updateNodeInfoOnElect(topo,req,activeNodes.cont[i],nd,&tmpNode->inNeighb.cont
1364         activateNodeInfo(topo,nd);
1365         activeFlags[nd]=1;
1366     }
1367 }
1368     activeFlags[activeNodes.cont[i]]=2;
1369 }
1370 else if (activeFlags[activeNodes.cont[i]]==0)
1371 {
1372     addError(CRITICAL,"Internal unconsistency in %s at line %d",
1373             __FILE__,__LINE__);
1374     longListEnd(&activeNodes);
1375     free(activeFlags);
1376     return -1;
1377 }
1378 }
1379 for (i=0;i<threshold;i++)
1380 {
1381     if (activeFlags[activeNodes.cont[i]]==2)
1382     {
1383         topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost=
1384         topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost;
1385         topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].neighbInd=
1386         topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd;
1387         activateNodeInfo(topo,activeNodes.cont[i]);
1388     }
1389 }
1390 }
1391
1392 longListEnd(&activeNodes);
1393 free(activeFlags);
1394
1395 #ifdef DEBUG
1396     printf("Bellman-Kalaba : %ld iterations \n",iter);
1397 #endif
1398
1399     return 0;
1400 }

```

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().

```

187 {
188     BKConnect *ptr=NULL;
189
190     if (dst == NULL || dst->cont == NULL ||
191         src == NULL || src->cont == NULL)
192     {
193         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
194                 __FILE__,__LINE__);
195         return -1;
196     }
197
198     if (dst->size < src->size)

```

```

199     {
200         if ((ptr=(BKConnect*) realloc(dst->cont,src->size*sizeof(BKConnect)))==NULL)
201         {
202             addError(CRITICAL,"Critical lack of memory in %s at line %d",
203                 __FILE__,__LINE__);
204             return -1;
205         }
206         else
207         {
208             dst->cont=ptr;
209             dst->size=src->size;
210         }
211     }
212
213     memcpy(dst->cont,src->cont,src->size*sizeof(BKConnect));
214     dst->top=src->top;
215
216     return 0;
217 }

```

4.23.1.4 int bkConnectVecDestroy (**BKConnectVec** * *vec*)

Definition at line 171 of file primaryPath.c.

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

```

172 {
173     if (vec == NULL || vec->cont == NULL)
174     {
175         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
176             __FILE__,__LINE__);
177         return -1;
178     }
179
180     free(vec->cont);
181     free(vec);
182
183     return 0;
184 }

```

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 BKConnectVec_::top.

Referenced by bkNodeVecDestroy(), bkNodeVecEnd(), bkNodeVecInit(), bkNodeVecNew(), and fill-Topo().

```

155 {
156     if (vec == NULL || vec->cont == NULL)
157     {
158         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
159             __FILE__,__LINE__);
160         return -1;
161     }
162
163     free(vec->cont);
164     vec->cont = NULL;
165     vec->size = 0;

```

```

166     vec->top = 0;
167
168     return 0;
169 }

```

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_::neighb-Id, and BKConnectVec_::size.

```

298 {
299     if (vec == NULL || vec->cont == NULL || val == NULL)
300     {
301         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
302                 __FILE__, __LINE__);
303         return -1;
304     }
305
306     if (index < 0)
307     {
308         addError(CRITICAL, "Bad argument (index < 0) in %s at line %d",
309                 __FILE__, __LINE__);
310         return -1;
311     }
312
313     if (index >= vec->size)
314     {
315         addError(CRITICAL, "Bad argument (wrong index) in %s at line %d",
316                 __FILE__, __LINE__);
317         return -1;
318     }
319
320     vec->cont[index].neighbId = val->neighbId;
321     vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!
322
323     return 0;
324 }

```

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().

```

127 {
128     BKConnect *ptr=NULL;
129
130     if (vec == NULL)
131     {
132         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
133                 __FILE__, __LINE__);
134         return -1;
135     }
136
137     if (size == -1)
138         size = BKCONNECTVEC_INITSIZE;
139

```

```

140     if ((ptr = (BKConnect*) calloc(size,sizeof(BKConnect))) == NULL)
141     {
142         addError(CRITICAL,"Critical lack of memory in %s at line %d",
143             __FILE__,__LINE__);
144         return -1;
145     }
146
147     vec->size = size;
148     vec->top = 0;
149     vec->cont = ptr;
150
151     return 0;
152 }

```

4.23.1.8 **BKConnectVec*** bkConnectVecNew (long size)

Definition at line 96 of file primaryPath.c.

```

97 {
98     BKConnectVec *vec=NULL;
99     BKConnect *ptr=NULL;
100
101     if ((vec = calloc(1,sizeof(BKConnectVec))) == NULL)
102     {
103         addError(CRITICAL,"Critical lack of memory in %s at line %d",
104             __FILE__,__LINE__);
105         return NULL;
106     }
107
108     if (size == -1)
109         size = BKCONNECTVEC_INITSIZE;
110
111     if ((ptr = (BKConnect*) calloc(size,sizeof(BKConnect))) == NULL)
112     {
113         addError(CRITICAL,"Critical lack of memory in %s at line %d",
114             __FILE__,__LINE__);
115         free(vec);
116         return NULL;
117     }
118
119     vec->size = size;
120     vec->top = 0;
121     vec->cont = ptr;
122
123     return vec;
124 }

```

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::neighb-Id`, and `BKConnectVec::top`.

```

251 {
252     if (vec == NULL || vec->cont == NULL || val == NULL)
253     {
254         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
255             __FILE__,__LINE__);
256         return -1;
257     }

```

```

258
259     if (vec->top == 0)
260     {
261         addError(CRITICAL,"Pop on empty stack in %s at line %d",
262             __FILE__,__LINE__);
263         return -1;
264     }
265
266     val->neighbId = vec->cont[vec->top - 1].neighbId;
267     val->linkState = vec->cont[vec->top--].linkState;
268
269     return 0;
270 }

```

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().

```

220 {
221     void* ptr=NULL;
222
223     if (vec == NULL || vec->cont == NULL)
224     {
225         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
226             __FILE__,__LINE__);
227         return -1;
228     }
229
230     if (vec->top >= vec->size)
231     {
232         if ((ptr = realloc(vec->cont, vec->size *
233             2 * sizeof(BKConnect))) == NULL)
234         {
235             addError(CRITICAL,"Critical lack of memory in %s at line %d",
236                 __FILE__,__LINE__);
237             return -1;
238         }
239
240         vec->size *= 2;
241         vec->cont = ptr;
242     }
243
244     vec->cont[vec->top].neighbId = val->neighbId;
245     vec->cont[vec->top++].linkState = val->linkState; // pointeur directement sur la DB!
246
247     return 0;
248 }

```

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().

```

273 {
274     void* ptr=NULL;
275
276     if (vec == NULL || vec->cont == NULL)
277     {
278         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
279                 __FILE__,__LINE__);
280         return -1;
281     }
282
283     if ((ptr = realloc(vec->cont, newsize*sizeof(BKConnect))) == NULL)
284     {
285         addError(CRITICAL,"Critical lack of memory in %s at line %d",
286                 __FILE__,__LINE__);
287         return -1;
288     }
289
290     vec->cont = ptr;
291     memset(ptr+ (vec->size * sizeof(BKConnect)), 0, (newsize - vec->size)*sizeof(BKConnect));
292     vec->size = newsize;
293
294     return 0;
295 }

```

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.

```

327 {
328     if (vec == NULL || vec->cont == NULL)
329     {
330         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
331                 __FILE__,__LINE__);
332         return -1;
333     }
334
335     if (index < 0)
336     {
337         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
338                 __FILE__,__LINE__);
339         return -1;
340     }
341
342     if (index >= vec->size)
343     {
344         if (bkConnectVecResize(vec,max(vec->size * 2,index+1))<0)
345         {
346             addError(CRITICAL,"Unable to resize vector in %s at line %d",
347                     __FILE__,__LINE__);
348             return -1;
349         }
350     }
351
352     vec->cont[index].neighbId = val->neighbId;
353     vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!
354     vec->top=max(vec->top,index+1);
355
356     return 0;
357 }

```

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.

```

510 {
511     long i;
512
513     if (vec == NULL || vec->cont == NULL)
514     {
515         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
516                 __FILE__, __LINE__);
517         return -1;
518     }
519
520     for (i=0; i<vec->size; i++)
521     {
522         bkConnectVecEnd(&vec->cont[i].inNeighb);
523         bkConnectVecEnd(&vec->cont[i].outNeighb);
524     }
525
526     free(vec->cont);
527     free(vec);
528
529     return 0;
530 }
```

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().

```

485 {
486     long i;
487
488     if (vec == NULL || vec->cont == NULL)
489     {
490         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
491                 __FILE__, __LINE__);
492         return -1;
493     }
494
495     for (i=0; i<vec->size; i++)
496     {
497         bkConnectVecEnd(&vec->cont[i].inNeighb);
498         bkConnectVecEnd(&vec->cont[i].outNeighb);
499     }
500
501     free(vec->cont);
502     vec->cont = NULL;
503     vec->size = 0;
504     vec->top = 0;
505
506     return 0;
507 }
```


4.23.1.15 BNode* bkNodeVecGet (BNodeVec * vec, long index)

Definition at line 640 of file primaryPath.c.

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

Referenced by bellmanKalaba(), printTopo(), and updateRequest().

```

641 {
642     if (vec == NULL || vec->cont == NULL)
643     {
644         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
645             __FILE__,__LINE__);
646         return NULL;
647     }
648
649     if (index < 0)
650     {
651         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
652             __FILE__,__LINE__);
653         return NULL;
654     }
655
656     if (index >= vec->size)
657     {
658         addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
659             __FILE__,__LINE__);
660         return NULL;
661     }
662
663     return vec->cont+index;
664 }
```

4.23.1.16 int bkNodeVecInit (BNodeVec * vec, long size)

Definition at line 426 of file primaryPath.c.

References addError(), bkConnectVecEnd(), bkConnectVecInit(), BKNODEVEC_INITSIZE, calloc, BNodeVec::cont, CRITICAL, free, BNodeVec::size, and BNodeVec::top.

Referenced by initTopo().

```

427 {
428     BNode* ptr=NULL;
429     long i,j;
430
431     if (vec == NULL)
432     {
433         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
434             __FILE__,__LINE__);
435         return -1;
436     }
437
438     if (size == -1)
439         size = BKNODEVEC_INITSIZE;
440
441     if ((ptr = calloc(size,sizeof(BNode))) == NULL)
442     {
443         addError(CRITICAL,"Critical lack of memory in %s at line %d",
444             __FILE__,__LINE__);
445         return -1;
446     }
447 }
```

```

448     for (i=0;i<size;i++)
449     {
450         if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)
451         {
452             for (j=i-1;j>=0;j--)
453             {
454                 bkConnectVecEnd(&ptr[j].inNeighb);
455                 bkConnectVecEnd(&ptr[j].outNeighb);
456             }
457             addError(CRITICAL,"Unable to initialize structure in %s at line %d",
458                     __FILE__,__LINE__);
459             free(ptr);
460             return -1;
461         }
462         else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
463         {
464             bkConnectVecEnd(&ptr[i].inNeighb);
465             for (j=i-1;j>=0;j--)
466             {
467                 bkConnectVecEnd(&ptr[j].inNeighb);
468                 bkConnectVecEnd(&ptr[j].outNeighb);
469             }
470             addError(CRITICAL,"Unable to initialize structure in %s at line %d",
471                     __FILE__,__LINE__);
472             free(ptr);
473             return -1;
474         }
475     }
476
477     vec->size = size;
478     vec->top = 0;
479     vec->cont = ptr;
480
481     return 0;
482 }

```

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.

```

365 {
366     BKNodeVec* vec=NULL;
367     BKNode* ptr=NULL;
368     long i,j;
369
370     if ((vec = calloc(1,sizeof(BKNodeVec))) == NULL)
371     {
372         addError(CRITICAL,"Critical lack of memory in %s at line %d",
373                 __FILE__,__LINE__);
374         return NULL;
375     }
376
377     if (size == -1)
378         size = BKNODEVEC_INITSIZE;
379
380     if ((ptr = calloc(size,sizeof(BKNode))) == NULL)
381     {
382         addError(CRITICAL,"Critical lack of memory in %s at line %d",
383                 __FILE__,__LINE__);
384         free(vec);
385         return NULL;

```

```

386     }
387
388     for (i=0;i<size;i++)
389     {
390         if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)
391         {
392             for (j=i-1;j>=0;j--)
393             {
394                 bkConnectVecEnd(&ptr[j].inNeighb);
395                 bkConnectVecEnd(&ptr[j].outNeighb);
396             }
397             addError(CRITICAL,"Unable to initialize structure in %s at line %d",
398                     __FILE__,__LINE__);
399             free(vec);
400             free(ptr);
401             return NULL;
402         }
403         else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
404         {
405             bkConnectVecEnd(&ptr[i].inNeighb);
406             for (j=i-1;j>=0;j--)
407             {
408                 bkConnectVecEnd(&ptr[j].inNeighb);
409                 bkConnectVecEnd(&ptr[j].outNeighb);
410             }
411             addError(CRITICAL,"Unable to initialize structure in %s at line %d",
412                     __FILE__,__LINE__);
413             free(vec);
414             free(ptr);
415             return NULL;
416         }
417     }
418
419     vec->size = size;
420     vec->top = 0;
421     vec->cont = ptr;
422
423     return vec;
424 }

```

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.

```

570 {
571     if (vec == NULL || vec->cont == NULL || val == NULL)
572     {
573         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
574                 __FILE__,__LINE__);
575         return -1;
576     }
577
578     if (vec->top == 0)
579     {
580         addError(CRITICAL,"Pop on empty stack in %s at line %d",
581                 __FILE__,__LINE__);
582         return -1;
583     }
584
585     if (bkConnectVecCopy(&val->inNeighb,&vec->cont[vec->top-1].inNeighb)<0)
586     {

```

```

587         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
588                 __FILE__,__LINE__);
589     return -1;
590 }
591 if (bkConnectVecCopy(&val->outNeighb,&vec->cont[vec->top-1].outNeighb)<0)
592 {
593     addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
594             __FILE__,__LINE__);
595     return -1;
596 }
597 val->nodeId = vec->cont[vec->top-1].nodeId;
598 val->neighbInd = vec->cont[vec->top--].neighbInd;
599
600 return 0;
601 }

```

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().

```

533 {
534     if (vec == NULL || vec->cont == NULL)
535     {
536         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
537                 __FILE__,__LINE__);
538         return -1;
539     }
540
541     if (vec->top >= vec->size)
542     {
543         if (bkNodeVecResize(vec,vec->size*2)<0)
544         {
545             addError(CRITICAL,"Critical lack of memory in %s at line %d",
546                     __FILE__,__LINE__);
547             return -1;
548         }
549     }
550
551     if (bkConnectVecCopy(&vec->cont[vec->top].inNeighb,&val->inNeighb)<0)
552     {
553         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
554                 __FILE__,__LINE__);
555         return -1;
556     }
557     if (bkConnectVecCopy(&vec->cont[vec->top].outNeighb,&val->outNeighb)<0)
558     {
559         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
560                 __FILE__,__LINE__);
561         return -1;
562     }
563     vec->cont[vec->top].nodeId = val->nodeId;
564     vec->cont[vec->top++].neighbInd = val->neighbInd;
565
566     return 0;
567 }

```

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().

```

604 {
605     void *ptr=NULL;
606     long i;
607
608     if (vec == NULL || vec->cont == NULL)
609     {
610         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
611             __FILE__,__LINE__);
612         return -1;
613     }
614
615     if ((ptr = (BKNode*) realloc(vec->cont, newsize*sizeof(BKNode))) == NULL)
616     {
617         addError(CRITICAL,"Critical lack of memory in %s at line %d",
618             __FILE__,__LINE__);
619         return -1;
620     }
621
622     memset(ptr+(vec->size*sizeof(BKNode)) , 0, (newsize-vec->size)*sizeof(BKNode));
623     vec->cont = ptr;
624
625     for (i=vec->size;i<newsize;i++)
626     {
627         if (bkConnectVecInit(&((BKNode*) ptr)[i].inNeighb,-1)<0 ||
628             bkConnectVecInit(&((BKNode*) ptr)[i].outNeighb,-1)<0)
629         {
630             addError(CRITICAL,"Unable to initialize structure in %s at line %d",
631                 __FILE__,__LINE__);
632             return -1;
633         }
634     }
635     vec->size = newsize;
636
637     return 0;
638 }
```

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.

```

667 {
668     if (vec == NULL || vec->cont == NULL)
669     {
670         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
671             __FILE__,__LINE__);
672         return -1;
673     }
674
675     if (index < 0)
676     {
```

```

677     addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
678             __FILE__,__LINE__);
679     return -1;
680 }
681
682 if (index >= vec->size)
683 {
684     if (bkNodeVecResize(vec,max(vec->size * 2,index+1))<0)
685     {
686         addError(CRITICAL,"Unable to resize node vector in %s at line %d",
687                 __FILE__,__LINE__);
688         return -1;
689     }
690 }
691
692 if (bkConnectVecCopy(&vec->cont[index].inNeighb,&val->inNeighb)<0)
693 {
694     addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
695             __FILE__,__LINE__);
696     return -1;
697 }
698 if (bkConnectVecCopy(&vec->cont[index].outNeighb,&val->outNeighb)<0)
699 {
700     addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
701             __FILE__,__LINE__);
702     return -1;
703 }
704 vec->cont[index].nodeId = val->nodeId;
705 vec->cont[index].neighbInd = val->neighbInd;
706 vec->top=max(vec->top,index+1);
707
708 return 0;
709 }

```

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().

```

22 {
23     BKTopology topo;
24     long src;
25
26 #if defined LINUX && defined TIME4
27     struct timezone tz;
28     struct timeval t1,t2;
29 #endif
30
31     if (dataBase == NULL || req==NULL)
32     {
33         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
34                 __FILE__,__LINE__);
35         return -1;
36     }

```

```

37
38 #if defined LINUX && defined TIMING && defined TIME4
39     gettimeofday(&t1, &tz);
40 #endif
41
42     if ((src=getRequestSrc(req))<0)
43     {
44         addError(CRITICAL,"Unable to get requested source in %s at line %d",
45             __FILE__,__LINE__);
46         return -1;
47     }
48
49     if (initTopo(&topo,-1)<0)
50     {
51         addError(CRITICAL,"Unable to initialize the topology structure in %s at line %d",
52             __FILE__,__LINE__);
53         return -1;
54     }
55
56     if (fillTopo(dataBase,req,&topo)<0)
57     {
58         addError(CRITICAL,"Unable to build topology in %s at line %d",
59             __FILE__,__LINE__);
60         endTopo(&topo);
61         return -1;
62     }
63     //printTopo(&topo);
64
65     if (bellmanKalaba(&topo,req)<0)
66     {
67         addError(CRITICAL,"Bellman-Kalaba failure in %s at line %d",
68             __FILE__,__LINE__);
69         endTopo(&topo);
70         return -1;
71     }
72
73     if (updateRequest(&topo,req)<0)
74     {
75         addError(CRITICAL,"Unable to update requested path in %s at line %d",
76             __FILE__,__LINE__);
77         endTopo(&topo);
78         return -1;
79     }
80
81 #if defined LINUX && defined TIMING && defined TIME4
82     gettimeofday(&t2, &tz);
83     fprintf(stderr, "Time for calculation of primary path : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
84         (t2.tv_usec - t1.tv_usec) / 1000.0);
85 #endif
86
87     endTopo(&topo);
88
89     return 0;
90 }

```

4.23.1.23 int endTopo (BKTopology * topo)

Definition at line 742 of file primaryPath.c.

References addError(), bkNodeVecEnd(), CRITICAL, longVecEnd(), BKTopology::nodeInd, and BKTopology::nodeVec.

Referenced by computePrimaryPath().

```

743 {

```

```

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",

```



```

794         __FILE__, __LINE__);
795     longListEnd(&toDoNodes);
796     return -1;
797 }
798
799 memset(&tmpNode, 0, sizeof(BKNode));
800 if (bkConnectVecInit(&tmpNode.inNeighb, -1) < 0)
801 {
802     addError(CRITICAL, "Unable to initialize the temporary node in %s at line %d",
803             __FILE__, __LINE__);
804     longListEnd(&toDoNodes);
805     free(activeFlags);
806     return -1;
807 }
808 if (bkConnectVecInit(&tmpNode.outNeighb, -1) < 0)
809 {
810     addError(CRITICAL, "Unable to initialize the temporary node in %s at line %d",
811             __FILE__, __LINE__);
812     longListEnd(&toDoNodes);
813     free(activeFlags);
814     bkConnectVecEnd(&tmpNode.inNeighb);
815     return -1;
816 }
817
818 if (longListPushBack(&toDoNodes, src) < 0)
819 {
820     addError(CRITICAL, "Unable to push back on list of longs in %s at line %d",
821             __FILE__, __LINE__);
822     longListEnd(&toDoNodes);
823     free(activeFlags);
824     bkConnectVecEnd(&tmpNode.inNeighb);
825     bkConnectVecEnd(&tmpNode.outNeighb);
826     return -1;
827 }
828 activeFlags[src]=1;
829 while (toDoNodes.top>0)
830 {
831     if (longListPopBack(&toDoNodes, &nd) < 0)
832     {
833         addError(CRITICAL, "Unable to pop back on list of longs in %s at line %d",
834                 __FILE__, __LINE__);
835         longListEnd(&toDoNodes);
836         free(activeFlags);
837         bkConnectVecEnd(&tmpNode.inNeighb);
838         bkConnectVecEnd(&tmpNode.outNeighb);
839         return -1;
840     }
841
842     tmpNode.inNeighb.top=0;
843     if ((tmpNeighb=DBgetNodeInNeighb(dataBase, nd)) == NULL)
844     {
845         addError(CRITICAL, "Unable to get the list of neighbours in %s at line %d",
846                 __FILE__, __LINE__);
847         longListEnd(&toDoNodes);
848         free(activeFlags);
849         bkConnectVecEnd(&tmpNode.inNeighb);
850         bkConnectVecEnd(&tmpNode.outNeighb);
851         return -1;
852     }
853     for (i=0; i<tmpNeighb->top; i++)
854     {
855         if (activeFlags[tmpNeighb->cont[i]]==2)
856         {
857             nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
858             for (j=0; (j<nodePtr->outNeighb.top) && (nodePtr->outNeighb.cont[j].neighbId!=nd); j++);
859             if (j<nodePtr->outNeighb.top)
860             {

```

```

861         tmpConn.neighbId=tmpNeighb->cont[i];
862         tmpConn.linkState=nodePtr->outNeighb.cont[j].linkState;
863         memset(&tmpConn.info,0,sizeof(BKConnectInfo));
864         memcpy(tmpConn.info.gain,nodePtr->outNeighb.cont[j].info.gain,NB_OA*sizeof(double));
865         if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)
866         {
867             addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
868                     __FILE__,__LINE__);
869             longListEnd(&toDoNodes);
870             free(activeFlags);
871             bkConnectVecEnd(&tmpNode.inNeighb);
872             bkConnectVecEnd(&tmpNode.outNeighb);
873             return -1;
874         }
875     }
876 }
877 else
878 {
879     tmpConn.neighbId=tmpNeighb->cont[i];
880     tmpConn.linkState=DBgetLinkState(dataBase,tmpNeighb->cont[i],nd);
881     memset(&tmpConn.info,0,sizeof(BKConnectInfo));
882     if (isValidRequestLink(dataBase,tmpNeighb->cont[i],nd,
883                           tmpConn.linkState,req,tmpConn.info.gain))
884     {
885         if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)
886         {
887             addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
888                     __FILE__,__LINE__);
889             longListEnd(&toDoNodes);
890             free(activeFlags);
891             bkConnectVecEnd(&tmpNode.inNeighb);
892             bkConnectVecEnd(&tmpNode.outNeighb);
893             return -1;
894         }
895     }
896 }
897 }
898
899 tmpNode.outNeighb.top=0;
900 if ((tmpNeighb=DBgetNodeOutNeighb(dataBase,nd))==NULL)
901 {
902     addError(CRITICAL,"Unable to get the list of neighbours in %s at line %d",
903             __FILE__,__LINE__);
904     longListEnd(&toDoNodes);
905     free(activeFlags);
906     bkConnectVecEnd(&tmpNode.inNeighb);
907     bkConnectVecEnd(&tmpNode.outNeighb);
908     return -1;
909 }
910 for (i=0;i<tmpNeighb->top;i++)
911 {
912     if (activeFlags[tmpNeighb->cont[i]]==2)
913     {
914         nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
915         for (j=0;(j<nodePtr->inNeighb.top) && (nodePtr->inNeighb.cont[j].neighbId!=nd);j++);
916         if (j<nodePtr->inNeighb.top)
917         {
918             tmpConn.neighbId=tmpNeighb->cont[i];
919             tmpConn.linkState=nodePtr->inNeighb.cont[j].linkState;
920             memset(&tmpConn.info,0,sizeof(BKConnectInfo));
921             memcpy(tmpConn.info.gain,nodePtr->inNeighb.cont[j].info.gain,NB_OA*sizeof(double));
922             if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
923             {
924                 addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
925                         __FILE__,__LINE__);
926                 longListEnd(&toDoNodes);
927                 free(activeFlags);

```

```

928             bkConnectVecEnd(&tmpNode.inNeighb);
929             bkConnectVecEnd(&tmpNode.outNeighb);
930             return -1;
931         }
932     }
933 }
934 else
935 {
936     tmpConn.neighbId=tmpNeighb->cont[i];
937     tmpConn.linkState=DBgetLinkState(dataBase,nd,tmpNeighb->cont[i]);
938     memset(&tmpConn.info,0,sizeof(BKConnectInfo));
939     if (isValidRequestLink(dataBase,nd,tmpNeighb->cont[i],
940                           tmpConn.linkState,req,tmpConn.info.gain))
941     {
942         if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
943         {
944             addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
945                     __FILE__,__LINE__);
946             longListEnd(&toDoNodes);
947             free(activeFlags);
948             bkConnectVecEnd(&tmpNode.inNeighb);
949             bkConnectVecEnd(&tmpNode.outNeighb);
950             return -1;
951         }
952     }
953 }
954 if (activeFlags[tmpNeighb->cont[i]]==0)
955 {
956     if (longListPushBack(&toDoNodes,tmpNeighb->cont[i])<0)
957     {
958         addError(CRITICAL,"Unable to push back on list of longs in %s at line %d",
959                 __FILE__,__LINE__);
960         longListEnd(&toDoNodes);
961         free(activeFlags);
962         return -1;
963     }
964     activeFlags[tmpNeighb->cont[i]]=1;
965 }
966 }
967
968 tmpNode.nodeId=nd;
969 tmpNode.neighbInd=-1;
970 if (bkNodeVecPushBack(&topo->nodeVec,&tmpNode)<0)
971 {
972     addError(CRITICAL,"Unable to push back node in %s at line %d",
973             __FILE__,__LINE__);
974     longListEnd(&toDoNodes);
975     free(activeFlags);
976     bkConnectVecEnd(&tmpNode.inNeighb);
977     bkConnectVecEnd(&tmpNode.outNeighb);
978     return -1;
979 }
980
981 if (longVecSet(&topo->nodeInd,nd,(topo->nodeVec.top-1))<0)
982 {
983     addError(CRITICAL,"Unable to set node index in %s at line %d",
984             __FILE__,__LINE__);
985     longListEnd(&toDoNodes);
986     free(activeFlags);
987     bkConnectVecEnd(&tmpNode.inNeighb);
988     bkConnectVecEnd(&tmpNode.outNeighb);
989     return -1;
990 }
991
992 activeFlags[nd]=2;
993 }
994

```

```

995     if (((topo->nbNodes=DBgetNbNodes(dataBase))<0)||
996         ((topo->nbLinks=DBgetNbLinks(dataBase))<0))
997     {
998         addError(CRITICAL,"Unable to get number of nodes and links in %s at line %d",
999             __FILE__,__LINE__);
1000         longListEnd(&toDoNodes);
1001         free(activeFlags);
1002         bkConnectVecEnd(&tmpNode.inNeighb);
1003         bkConnectVecEnd(&tmpNode.outNeighb);
1004         return -1;
1005     }
1006
1007     longListEnd(&toDoNodes);
1008     free(activeFlags);
1009     bkConnectVecEnd(&tmpNode.inNeighb);
1010     bkConnectVecEnd(&tmpNode.outNeighb);
1011
1012     return 0;
1013 }

```

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().

```

1078 {
1079     if (req==NULL)
1080     {
1081         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1082             __FILE__,__LINE__);
1083         return -1;
1084     }
1085
1086     if (req->path.top<2 || req->path.cont[0]<0 ||
1087         req->path.cont[req->path.top-1]<0)
1088     {
1089         addError(CRITICAL,"Bad requested path format in %s at line %d",
1090             __FILE__,__LINE__);
1091         return -1;
1092     }
1093
1094     return req->path.cont[req->path.top-1];
1095 }

```

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().

```

1058 {
1059     if (req==NULL)
1060     {
1061         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1062             __FILE__,__LINE__);
1063         return -1;

```

```

1064     }
1065
1066     if (req->path.top<2 || req->path.cont[0]<0 ||
1067         req->path.cont[req->path.top-1]<0)
1068     {
1069         addError(CRITICAL,"Bad requested path format in %s at line %d",
1070                 __FILE__,__LINE__);
1071         return -1;
1072     }
1073
1074     return req->path.cont[0];
1075 }

```

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().

```

1403 {
1404     bool process=FALSE;
1405     long i,j,k,l,top;
1406     double tmpSum;
1407
1408     if (topo == NULL)
1409     {
1410         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1411                 __FILE__,__LINE__);
1412         return -1;
1413     }
1414
1415     for (i=0;i<NB_OA;i++)
1416     {
1417         if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
1418         {
1419             process=TRUE;
1420         }
1421     }
1422
1423     if (process)
1424     {
1425         for (k=0;k<NB_OA;k++)
1426         {
1427             topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]=0;
1428         }
1429         for (i=0;i<topo->nodeVec.top;i++)
1430         {
1431             top=topo->nodeVec.cont[i].inNeighb.top;
1432             for (j=0;j<top;j++)
1433             {
1434                 for (k=0;k<NB_OA;k++)
1435                 {
1436                     tmpSum=0;
1437                     for (l=0;l<NB_PREEMPTION;l++)
1438                     {
1439                         tmpSum+=topo->nodeVec.cont[i].inNeighb.cont[j].linkState->pbw[k][l];
1440                     }

```

```

1441             topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]+=tmpSum/
1442             topo->nodeVec.cont[i].inNeighb.cont[j].linkState->cap[k];
1443         }
1444     }
1445 }
1446 }
1447
1448     return 0;
1449 }

```

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().

```

716 {
717     if (topo == NULL)
718     {
719         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
720             __FILE__,__LINE__);
721         return -1;
722     }
723
724     if (bkNodeVecInit(&topo->nodeVec,-1)<0)
725     {
726         addError(CRITICAL,"Unable to initialize node vector in %s at line %d",
727             __FILE__,__LINE__);
728         return -1;
729     }
730
731     if (longVecInit(&topo->nodeInd,size)<0)
732     {
733         addError(CRITICAL,"Unable to initialize long vector in %s at line %d",
734             __FILE__,__LINE__);
735         bkNodeVecEnd(&topo->nodeVec);
736         return -1;
737     }
738
739     return 0;
740 }

```

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, PrimaryComputationConfig_::relLoad, PrimaryComputationConfig_::rerouting, PrimaryComputationConfig_::sqLoad, and PrimaryComputationConfig_::sqRelLoad.

Referenced by bellmanKalaba().

```

1453 {
1454     double score=0,totBW[NB_OA],newSum,rerouteScore=0;
1455     long i,j;
1456
1457     if (topo == NULL || connect == NULL)
1458     {
1459         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1460             __FILE__,__LINE__);
1461         return HUGE_VAL;
1462     }
1463
1464     score=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.cost;
1465
1466     for (i=0;i<NB_OA;i++)
1467     {
1468         totBW[i]=0;
1469         for (j=0;j<NB_PREEMPTION;j++)
1470         {
1471             totBW[i]+=connect->linkState->pbw[i][j];
1472         }
1473
1474         if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
1475         {
1476             score+=damoteConfig.primaryComputationConfig.loadBal[i]
1477                 *((totBW[i]+req->bw[i])/connect->linkState->cap[i])
1478                 *((totBW[i]+req->bw[i])/connect->linkState->cap[i]);
1479             score+=damoteConfig.primaryComputationConfig.loadBal[i]
1480                 *(totBW[i]/connect->linkState->cap[i])
1481                 *(totBW[i]/connect->linkState->cap[i]);
1482
1483             newSum=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkS
1484             if (!__isinf(newSum))
1485             {
1486                 return HUGE_VAL;
1487             }
1488
1489             score+=damoteConfig.primaryComputationConfig.loadBal[i]
1490                 *(-1/(double)topo->nbLinks)*newSum*newSum;
1491             score+=damoteConfig.primaryComputationConfig.loadBal[i]
1492                 *(1/(double)topo->nbLinks)*topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]
1493                 *topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i];
1494         }
1495         if (damoteConfig.primaryComputationConfig.load[i]!=0)
1496         {
1497             score+=damoteConfig.primaryComputationConfig.load[i]*req->bw[i];
1498         }
1499         if (damoteConfig.primaryComputationConfig.sqload[i]!=0)
1500         {
1501             score+=damoteConfig.primaryComputationConfig.sqload[i]
1502                 *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i]);
1503         }
1504         if (damoteConfig.primaryComputationConfig.reload[i]!=0)
1505         {
1506             score+=damoteConfig.primaryComputationConfig.reload[i]
1507                 *req->bw[i]/connect->linkState->cap[i];
1508         }
1509         if (damoteConfig.primaryComputationConfig.sqreload[i]!=0)
1510         {
1511             score+=damoteConfig.primaryComputationConfig.sqreload[i]
1512                 *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i])
1513                 /(connect->linkState->cap[i]*connect->linkState->cap[i]);
1514         }
1515         if (damoteConfig.primaryComputationConfig.delay[i]!=0)
1516         {
1517             score+=damoteConfig.primaryComputationConfig.delay[i]
1518                 *((1/(connect->linkState->cap[i]-totBW[i]-req->bw[i]))
1519                 -(1/(connect->linkState->cap[i]-totBW[i])));

```

```

1520     }
1521 }
1522
1523 for (i=0;i<NB_OA;i++)
1524 {
1525     if (damoteConfig.primaryComputationConfig.rerouting[i]!=0)
1526     {
1527         rerouteScore+=damoteConfig.primaryComputationConfig.rerouting[i]*
1528             makeRerouteScore(req,connect->info.gain,connect->linkState,i);
1529     }
1530 }
1531
1532 score+=rerouteScore*(score>0?1:0)*score;
1533
1534 return score;
1535 }

```

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().

```

1587 {
1588     BKNode* tmpNode;
1589
1590
1591     tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[src]];
1592     while (tmpNode->neighbInd!=-1 && tmpNode->nodeId!=dst)
1593     {
1594         tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[tmpNode->inNeighb.cont[tmpNode->info.newNeighbInd]];
1595     }
1596
1597     if (tmpNode->nodeId==dst)
1598         return 0;
1599
1600     return 1;
1601 }

```

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.

```

1016 {
1017     BKNode *tmpNode;
1018     long i,j;
1019
1020     for (i=0;i<topo->nodeInd.top;i++)
1021     {
1022         tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[i]);
1023         if (tmpNode!=NULL)
1024         {

```



```

1025         if (i!=tmpNode->nodeId)
1026         {
1027             addError(CRITICAL,"Topology unconsistency in %s at line %d",
1028                 __FILE__,__LINE__);
1029             return -1;
1030         }
1031
1032         printf("Node %ld\n-----\n",i);
1033         printf("Incoming neighbors : \n");
1034
1035         for (j=0; j<tmpNode->inNeighb.top; j++)
1036         {
1037             printf("%ld ", tmpNode->inNeighb.cont[j].neighbId);
1038         }
1039
1040         printf("\nOutgoing neighbors : \n");
1041
1042         for (j=0; j<tmpNode->outNeighb.top; j++)
1043         {
1044             printf("%ld ", tmpNode->outNeighb.cont[j].neighbId);
1045         }
1046         printf("\n");
1047
1048         printf("Chosen Neighbour Index: %ld \n",tmpNode->neighbInd);
1049
1050         printf("\n");
1051     }
1052 }
1053
1054 return 0;
1055 }

```

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().

```

1539 {
1540     long i;
1541
1542     if (topo == NULL || connect == NULL)
1543     {
1544         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1545             __FILE__,__LINE__);
1546         return -1;
1547     }
1548
1549     for (i=0;i<NB_OA;i++)
1550     {
1551         if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
1552         {
1553             topo->nodeVec.cont[topo->nodeInd.cont[dst]].info.newSum[i]=
1554                 topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkStat
1555             }
1556         }
1557
1558     return 0;
1559 }

```

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().

```

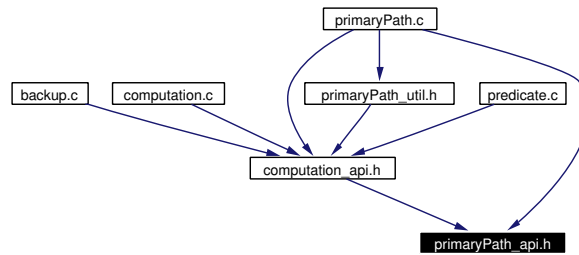
1098 {
1099     BKNode *tmpNode;
1100     long i,src,dst,nd;
1101
1102     if (topo == NULL || req==NULL)
1103     {
1104         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1105             __FILE__,__LINE__);
1106         return -1;
1107     }
1108
1109     if ((src=getRequestSrc(req))<0)
1110     {
1111         addError(CRITICAL,"Unable to get requested source in %s at line %d",
1112             __FILE__,__LINE__);
1113         return -1;
1114     }
1115
1116     if ((dst=getRequestDst(req))<0)
1117     {
1118         addError(CRITICAL,"Unable to get requested source in %s at line %d",
1119             __FILE__,__LINE__);
1120         return -1;
1121     }
1122
1123     req->path.top=0;
1124     nd=dst;
1125     if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
1126     {
1127         addError(CRITICAL,"Undetermined error in %s at line %d",
1128             __FILE__,__LINE__);
1129         return -1;
1130     }
1131     while (nd!=src)
1132     {
1133         if (tmpNode->neighbInd < 0)
1134         {
1135             addError(CRITICAL,"Destination unreachable in %s at line %d",
1136                 __FILE__,__LINE__);
1137             return -1;
1138         }
1139         if (longListPushBack(&req->path,nd)<0)
1140         {
1141             addError(CRITICAL,"Undetermined error in %s at line %d",
1142                 __FILE__,__LINE__);
1143             return -1;
1144         }
1145         nd=tmpNode->inNeighb.cont[tmpNode->neighbInd].neighbId;
1146         if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
1147         {
1148             addError(CRITICAL,"Undetermined error in %s at line %d",
1149                 __FILE__,__LINE__);
1150             return -1;
1151         }
1152     }
1153     if (longListPushBack(&req->path,nd)<0)

```

```
1154     {
1155         addError(CRITICAL,"Undetermined error in %s at line %d",
1156                 __FILE__,__LINE__);
1157         return -1;
1158     }
1159
1160     for (i=0;i<req->path.top/2;i++)
1161     {
1162         nd=req->path.cont[i];
1163         req->path.cont[i]=req->path.cont[req->path.top-1-i];
1164         req->path.cont[req->path.top-1-i]=nd;
1165     }
1166
1167     return 0;
1168 }
```

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()`.

```

22 {
23     BKTopology topo;
24     long src;
25
26 #if defined LINUX && defined TIME4
27     struct timezone tz;
28     struct timeval t1,t2;
29 #endif
30
31     if (dataBase == NULL || req==NULL)
32     {
33         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
34             __FILE__,__LINE__);
35         return -1;
36     }
37
38 #if defined LINUX && defined TIMING && defined TIME4
39     gettimeofday(&t1, &tz);
40 #endif
41

```

```

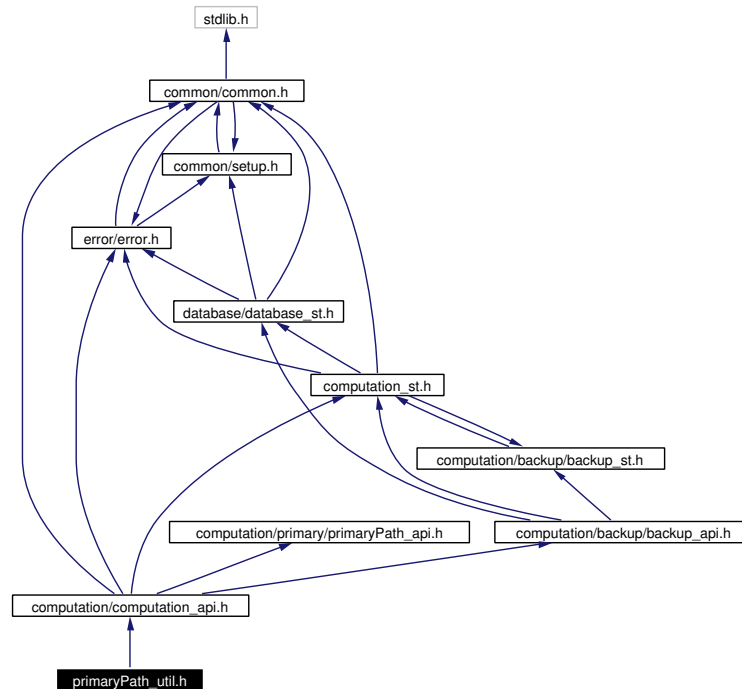
42     if ((src=getRequestSrc(req))<0)
43     {
44         addError(CRITICAL,"Unable to get requested source in %s at line %d",
45             __FILE__,__LINE__);
46         return -1;
47     }
48
49     if (initTopo(&topo,-1)<0)
50     {
51         addError(CRITICAL,"Unable to initialize the topology structure in %s at line %d",
52             __FILE__,__LINE__);
53         return -1;
54     }
55
56     if (fillTopo(dataBase,req,&topo)<0)
57     {
58         addError(CRITICAL,"Unable to build topology in %s at line %d",
59             __FILE__,__LINE__);
60         endTopo(&topo);
61         return -1;
62     }
63     //printTopo(&topo);
64
65     if (bellmanKalaba(&topo,req)<0)
66     {
67         addError(CRITICAL,"Bellman-Kalaba failure in %s at line %d",
68             __FILE__,__LINE__);
69         endTopo(&topo);
70         return -1;
71     }
72
73     if (updateRequest(&topo,req)<0)
74     {
75         addError(CRITICAL,"Unable to update requested path in %s at line %d",
76             __FILE__,__LINE__);
77         endTopo(&topo);
78         return -1;
79     }
80
81 #if defined LINUX && defined TIMING && defined TIME4
82     gettimeofday(&t2, &tz);
83     fprintf(stderr, "Time for calculation of primary path : %f ms\n", (t2.tv_sec - t1.tv_sec) * 1000 +
84         (t2.tv_usec - t1.tv_usec) / 1000.0);
85 #endif
86
87     endTopo(&topo);
88
89     return 0;
90 }

```

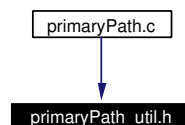
4.25 primaryPath_util.h File Reference

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

Include dependency graph for 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()`.

```

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 }
```

4.25.2.2 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().

```

1172 {
1173     LongList activeNodes;
1174     BKNode *tmpNode;
1175     bool done=FALSE;
1176     int *activeFlags;
1177     long src,i,j,k,nd,top,threshold,size,iter=0;
1178     double tmpCost;
1179
1180
1181     if (topo == NULL)
1182     {
1183         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1184             __FILE__,__LINE__);
1185         return -1;
1186     }
1187
1188     if ((src=getRequestSrc(req))<0)
1189     {
1190         addError(CRITICAL,"Unable to get requested source in %s at line %d",
1191             __FILE__,__LINE__);
1192         return -1;
1193     }
1194
1195     size=topo->nodeInd.top;
1196
1197     if (longListInit(&activeNodes,size)<0)
1198     {
1199         addError(CRITICAL,"Unable to initialize the active nodes list in %s at line %d",
1200             __FILE__,__LINE__);
1201         return -1;
1202     }
1203
1204     if ((activeFlags = (int*) calloc(size,sizeof(long))) == NULL)
1205     {
1206         addError(CRITICAL,"Critical lack of memory in %s at line %d",
1207             __FILE__,__LINE__);
1208         longListEnd(&activeNodes);
1209         return -1;
1210     }
1211
1212     if (src>=size)
1213     {
1214         addError(CRITICAL,"Inexistent node in %s at line %d",
1215             __FILE__,__LINE__);
1216         longListEnd(&activeNodes);
1217         free(activeFlags);
1218         return -1;
1219     }
1220
1221     if (initScore(src,topo)<0)
1222     {
1223         addError(CRITICAL,"Unable to initialize scores in %s at line %d",
1224             __FILE__,__LINE__);
1225         longListEnd(&activeNodes);
1226         free(activeFlags);
1227         return -1;
1228     }
1229     top=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.top;
1230     for (i=0;i<top;i++)
1231     {
1232         nd=topo->nodeVec.cont[topo->nodeInd.cont[src]].outNeighb.cont[i].neighbId;
1233         if (nd>=size)
1234         {

```

```

1234         addError(CRITICAL,"Inexistent node in %s at line %d",
1235                 __FILE__,__LINE__);
1236         longListEnd(&activeNodes);
1237         free(activeFlags);
1238         return -1;
1239     }
1240
1241     if (longListPushBack(&activeNodes,nd)<0)
1242     {
1243         addError(CRITICAL,"Undetermined error in %s at line %d",
1244                 __FILE__,__LINE__);
1245         longListEnd(&activeNodes);
1246         free(activeFlags);
1247         return -1;
1248     }
1249
1250     if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
1251     {
1252         addError(CRITICAL,"Undetermined error in %s at line %d",
1253                 __FILE__,__LINE__);
1254         longListEnd(&activeNodes);
1255         free(activeFlags);
1256         return -1;
1257     }
1258     for (k=0;(k<tmpNode->inNeighb.top) && (tmpNode->inNeighb.cont[k].neighbId!=src);k++);
1259     if (k>=tmpNode->inNeighb.top)
1260     {
1261         addError(CRITICAL,"Topology unconsistency in %s at line %d",
1262                 __FILE__,__LINE__);
1263         longListEnd(&activeNodes);
1264         free(activeFlags);
1265         return -1;
1266     }
1267     tmpNode->info.cost=makeScore(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
1268     tmpNode->info.newCost=tmpNode->info.cost;
1269     tmpNode->neighbInd=k;
1270     tmpNode->info.newNeighbInd=tmpNode->neighbInd;
1271     updateNodeInfoOnElect(topo,req,src,nd,&tmpNode->inNeighb.cont[k]);
1272     activateNodeInfo(topo,nd);
1273     activeFlags[nd]=1;
1274 }
1275 activeFlags[src]=2;
1276
1277 while (!done)
1278 {
1279     iter++;
1280     done=TRUE;
1281     threshold=activeNodes.top;
1282     for (i=0;i<threshold;i++)
1283     {
1284         top=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.top;
1285         for (j=0;j<top;j++)
1286         {
1287             nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.cont[j].neighbId;
1288             if (nd>=size)
1289             {
1290                 addError(CRITICAL,"Inexistent node in %s at line %d",
1291                         __FILE__,__LINE__);
1292                 longListEnd(&activeNodes);
1293                 free(activeFlags);
1294                 return -1;
1295             }
1296
1297             if (activeFlags[nd]!=0 && noLoop(topo,nd,activeNodes.cont[i]))
1298             {
1299                 tmpCost=makeScore(topo,req,nd,activeNodes.cont[i],
1300                                 &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].inNeighb.cont[j].neighbId);

```

```

1301         if (tmpCost=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost
1302         {
1303             done=FALSE;
1304             topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost=tmpC
1305             topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd
1306             updateNodeInfoOnElect(topo, req, nd, activeNodes.cont[i],
1307                                   &topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont
1308         }
1309     }
1310 }
1311
1312
1313 if (activeFlags[activeNodes.cont[i]]==1)
1314 {
1315     top=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.top;
1316     for (j=0;j<top;j++)
1317     {
1318         nd=topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].outNeighb.cont[j].
1319         if (nd>=size)
1320         {
1321             addError(CRITICAL, "Inexistent node in %s at line %d",
1322                     __FILE__, __LINE__);
1323             longListEnd(&activeNodes);
1324             free(activeFlags);
1325             return -1;
1326         }
1327
1328         if (activeFlags[nd]==0)
1329         {
1330             done=FALSE;
1331
1332             if (longListPushBack(&activeNodes, nd)<0)
1333             {
1334                 addError(CRITICAL, "Undetermined error in %s at line %d",
1335                         __FILE__, __LINE__);
1336                 longListEnd(&activeNodes);
1337                 free(activeFlags);
1338                 return -1;
1339             }
1340
1341             if ((tmpNode=bkNodeVecGet(&topo->nodeVec, topo->nodeInd.cont[nd]))==NULL)
1342             {
1343                 addError(CRITICAL, "Undetermined error in %s at line %d",
1344                         __FILE__, __LINE__);
1345                 longListEnd(&activeNodes);
1346                 free(activeFlags);
1347                 return -1;
1348             }
1349             for (k=0;(k<tmpNode->inNeighb.top) &&
1350                  (tmpNode->inNeighb.cont[k].neighbId!=activeNodes.cont[i]);k++);
1351             if (k>=tmpNode->inNeighb.top)
1352             {
1353                 addError(CRITICAL, "Topology unconsistency in %s at line %d",
1354                         __FILE__, __LINE__);
1355                 longListEnd(&activeNodes);
1356                 free(activeFlags);
1357                 return -1;
1358             }
1359             tmpNode->info.cost=makeScore(topo, req, activeNodes.cont[i], nd, &tmpNode->inNeig
1360             tmpNode->info.newCost=tmpNode->info.cost;
1361             tmpNode->neighbInd=k;
1362             tmpNode->info.newNeighbInd=tmpNode->neighbInd;
1363             updateNodeInfoOnElect(topo, req, activeNodes.cont[i], nd, &tmpNode->inNeighb.cont
1364             activateNodeInfo(topo, nd);
1365             activeFlags[nd]=1;
1366         }
1367     }

```

```

1368         activeFlags[activeNodes.cont[i]]=2;
1369     }
1370     else if (activeFlags[activeNodes.cont[i]]==0)
1371     {
1372         addError(CRITICAL,"Internal unconsistency in %s at line %d",
1373             __FILE__,__LINE__);
1374         longListEnd(&activeNodes);
1375         free(activeFlags);
1376         return -1;
1377     }
1378 }
1379 for (i=0;i<threshold;i++)
1380 {
1381     if (activeFlags[activeNodes.cont[i]]==2)
1382     {
1383         topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.cost=
1384             topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newCost;
1385         topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].neighbInd=
1386             topo->nodeVec.cont[topo->nodeInd.cont[activeNodes.cont[i]]].info.newNeighbInd;
1387         activateNodeInfo(topo,activeNodes.cont[i]);
1388     }
1389 }
1390 }
1391
1392 longListEnd(&activeNodes);
1393 free(activeFlags);
1394
1395 #ifdef DEBUG
1396     printf("Bellman-Kalaba : %ld iterations \n",iter);
1397 #endif
1398
1399     return 0;
1400 }

```

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().

```

187 {
188     BKConnect *ptr=NULL;
189
190     if (dst == NULL || dst->cont == NULL ||
191         src == NULL || src->cont == NULL)
192     {
193         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
194             __FILE__,__LINE__);
195         return -1;
196     }
197
198     if (dst->size < src->size)
199     {
200         if ((ptr=(BKConnect*) realloc(dst->cont,src->size*sizeof(BKConnect)))==NULL)
201         {
202             addError(CRITICAL,"Critical lack of memory in %s at line %d",
203                 __FILE__,__LINE__);
204             return -1;
205         }
206         else
207         {

```

```
208         dst->cont=ptr;
209         dst->size=src->size;
210     }
211 }
212
213 memcpy(dst->cont,src->cont,src->size*sizeof(BKConnect));
214 dst->top=src->top;
215
216 return 0;
217 }
```

4.25.2.4 int bkConnectVecDestroy ([BKConnectVec](#) *)

Definition at line 171 of file primaryPath.c.

References [addError\(\)](#), [BKConnectVec::cont](#), [CRITICAL](#), and [free](#).

```
172 {
173     if (vec == NULL || vec->cont == NULL)
174     {
175         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
176                 __FILE__,__LINE__);
177         return -1;
178     }
179
180     free(vec->cont);
181     free(vec);
182
183     return 0;
184 }
```

4.25.2.5 int bkConnectVecEnd ([BKConnectVec](#) *)

Definition at line 154 of file primaryPath.c.

References [addError\(\)](#), [BKConnectVec::cont](#), [CRITICAL](#), [free](#), [BKConnectVec::size](#), and [BKConnectVec::top](#).

Referenced by [bkNodeVecDestroy\(\)](#), [bkNodeVecEnd\(\)](#), [bkNodeVecInit\(\)](#), [bkNodeVecNew\(\)](#), and [fillTopo\(\)](#).

```
155 {
156     if (vec == NULL || vec->cont == NULL)
157     {
158         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
159                 __FILE__,__LINE__);
160         return -1;
161     }
162
163     free(vec->cont);
164     vec->cont = NULL;
165     vec->size = 0;
166     vec->top = 0;
167
168     return 0;
169 }
```

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.

```

298 {
299     if (vec == NULL || vec->cont == NULL || val == NULL)
300     {
301         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
302                 __FILE__, __LINE__);
303         return -1;
304     }
305
306     if (index < 0)
307     {
308         addError(CRITICAL, "Bad argument (index < 0) in %s at line %d",
309                 __FILE__, __LINE__);
310         return -1;
311     }
312
313     if (index >= vec->size)
314     {
315         addError(CRITICAL, "Bad argument (wrong index) in %s at line %d",
316                 __FILE__, __LINE__);
317         return -1;
318     }
319
320     vec->cont[index].neighbId = val->neighbId;
321     vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!
322
323     return 0;
324 }
```

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().

```

127 {
128     BKConnect *ptr=NULL;
129
130     if (vec == NULL)
131     {
132         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
133                 __FILE__, __LINE__);
134         return -1;
135     }
136
137     if (size == -1)
138         size = BKCONNECTVEC_INITSIZE;
139
140     if ((ptr = (BKConnect*) calloc(size, sizeof(BKConnect))) == NULL)
141     {
142         addError(CRITICAL, "Critical lack of memory in %s at line %d",
143                 __FILE__, __LINE__);
144         return -1;
145     }
146 }
```

```

147     vec->size = size;
148     vec->top = 0;
149     vec->cont = ptr;
150
151     return 0;
152 }

```

4.25.2.8 [BKConnectVec*](#) [bkConnectVecNew](#) (long)

Definition at line 96 of file primaryPath.c.

```

97 {
98     BKConnectVec *vec=NULL;
99     BKConnect *ptr=NULL;
100
101     if ((vec = calloc(1,sizeof(BKConnectVec))) == NULL)
102     {
103         addError(CRITICAL,"Critical lack of memory in %s at line %d",
104             __FILE__,__LINE__);
105         return NULL;
106     }
107
108     if (size == -1)
109         size = BKCONNECTVEC_INITSIZE;
110
111     if ((ptr = (BKConnect*) calloc(size,sizeof(BKConnect))) == NULL)
112     {
113         addError(CRITICAL,"Critical lack of memory in %s at line %d",
114             __FILE__,__LINE__);
115         free(vec);
116         return NULL;
117     }
118
119     vec->size = size;
120     vec->top = 0;
121     vec->cont = ptr;
122
123     return vec;
124 }

```

4.25.2.9 [int](#) [bkConnectVecPopBack](#) ([BKConnectVec *](#), [BKConnect *](#))

Definition at line 250 of file primaryPath.c.

References [addError\(\)](#), [BKConnectVec::cont](#), [CRITICAL](#), [BKConnect::linkState](#), [BKConnect::neighb-Id](#), and [BKConnectVec::top](#).

```

251 {
252     if (vec == NULL || vec->cont == NULL || val == NULL)
253     {
254         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
255             __FILE__,__LINE__);
256         return -1;
257     }
258
259     if (vec->top == 0)
260     {
261         addError(CRITICAL,"Pop on empty stack in %s at line %d",
262             __FILE__,__LINE__);
263         return -1;
264     }

```

```

265
266     val->neighbId = vec->cont[vec->top - 1].neighbId;
267     val->linkState = vec->cont[vec->top--].linkState;
268
269     return 0;
270 }

```

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().

```

220 {
221     void* ptr=NULL;
222
223     if (vec == NULL || vec->cont == NULL)
224     {
225         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
226                 __FILE__,__LINE__);
227         return -1;
228     }
229
230     if (vec->top >= vec->size)
231     {
232         if ((ptr = realloc(vec->cont, vec->size *
233                          2 * sizeof(BKConnect))) == NULL)
234         {
235             addError(CRITICAL,"Critical lack of memory in %s at line %d",
236                     __FILE__,__LINE__);
237             return -1;
238         }
239
240         vec->size *= 2;
241         vec->cont = ptr;
242     }
243
244     vec->cont[vec->top].neighbId = val->neighbId;
245     vec->cont[vec->top++].linkState = val->linkState; // pointeur directement sur la DB!
246
247     return 0;
248 }

```

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().

```

273 {
274     void* ptr=NULL;
275
276     if (vec == NULL || vec->cont == NULL)
277     {
278         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",

```



```

279     __FILE__, __LINE__);
280     return -1;
281 }
282
283 if ((ptr = realloc(vec->cont, newsize*sizeof(BKConnect))) == NULL)
284 {
285     addError(CRITICAL, "Critical lack of memory in %s at line %d",
286             __FILE__, __LINE__);
287     return -1;
288 }
289
290 vec->cont = ptr;
291 memset(ptr+ (vec->size * sizeof(BKConnect)), 0, (newsize - vec->size)*sizeof(BKConnect));
292 vec->size = newsize;
293
294 return 0;
295 }

```

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](#).

```

327 {
328     if (vec == NULL || vec->cont == NULL)
329     {
330         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
331                 __FILE__, __LINE__);
332         return -1;
333     }
334
335     if (index < 0)
336     {
337         addError(CRITICAL, "Bad argument (index < 0) in %s at line %d",
338                 __FILE__, __LINE__);
339         return -1;
340     }
341
342     if (index >= vec->size)
343     {
344         if (bkConnectVecResize(vec, max(vec->size * 2, index+1)) < 0)
345         {
346             addError(CRITICAL, "Unable to resize vector in %s at line %d",
347                     __FILE__, __LINE__);
348             return -1;
349         }
350     }
351
352     vec->cont[index].neighbId = val->neighbId;
353     vec->cont[index].linkState = val->linkState; // pointeur directement sur la DB!
354     vec->top = max(vec->top, index+1);
355
356     return 0;
357 }

```

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.

```

510 {
511     long i;
512
513     if (vec == NULL || vec->cont == NULL)
514     {
515         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
516                 __FILE__, __LINE__);
517         return -1;
518     }
519
520     for (i=0; i<vec->size; i++)
521     {
522         bkConnectVecEnd(&vec->cont[i].inNeighb);
523         bkConnectVecEnd(&vec->cont[i].outNeighb);
524     }
525
526     free(vec->cont);
527     free(vec);
528
529     return 0;
530 }
```

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().

```

485 {
486     long i;
487
488     if (vec == NULL || vec->cont == NULL)
489     {
490         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
491                 __FILE__, __LINE__);
492         return -1;
493     }
494
495     for (i=0; i<vec->size; i++)
496     {
497         bkConnectVecEnd(&vec->cont[i].inNeighb);
498         bkConnectVecEnd(&vec->cont[i].outNeighb);
499     }
500
501     free(vec->cont);
502     vec->cont = NULL;
503     vec->size = 0;
504     vec->top = 0;
505
506     return 0;
507 }
```

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().

```

641 {
642     if (vec == NULL || vec->cont == NULL)
643     {
644         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
645                 __FILE__,__LINE__);
646         return NULL;
647     }
648
649     if (index < 0)
650     {
651         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
652                 __FILE__,__LINE__);
653         return NULL;
654     }
655
656     if (index >= vec->size)
657     {
658         addError(CRITICAL,"Bad argument (wrong index) in %s at line %d",
659                 __FILE__,__LINE__);
660         return NULL;
661     }
662
663     return vec->cont+index;
664 }
```

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().

```

427 {
428     BKNode* ptr=NULL;
429     long i,j;
430
431     if (vec == NULL)
432     {
433         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
434                 __FILE__,__LINE__);
435         return -1;
436     }
437
438     if (size == -1)
439         size = BKNODEVEC_INITSIZE;
440
441     if ((ptr = calloc(size,sizeof(BKNode))) == NULL)
442     {
443         addError(CRITICAL,"Critical lack of memory in %s at line %d",
444                 __FILE__,__LINE__);
445         return -1;
446     }
447
448     for (i=0;i<size;i++)
449     {
450         if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)
451         {
452             for (j=i-1;j>=0;j--)
```

```

453         {
454             bkConnectVecEnd(&ptr[j].inNeighb);
455             bkConnectVecEnd(&ptr[j].outNeighb);
456         }
457         addError(CRITICAL,"Unable to initialize structure in %s at line %d",
458             __FILE__,__LINE__);
459         free(ptr);
460         return -1;
461     }
462     else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
463     {
464         bkConnectVecEnd(&ptr[i].inNeighb);
465         for (j=i-1;j>=0;j--)
466         {
467             bkConnectVecEnd(&ptr[j].inNeighb);
468             bkConnectVecEnd(&ptr[j].outNeighb);
469         }
470         addError(CRITICAL,"Unable to initialize structure in %s at line %d",
471             __FILE__,__LINE__);
472         free(ptr);
473         return -1;
474     }
475 }
476
477 vec->size = size;
478 vec->top = 0;
479 vec->cont = ptr;
480
481 return 0;
482 }

```

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.

```

365 {
366     BKNodeVec* vec=NULL;
367     BKNode* ptr=NULL;
368     long i,j;
369
370     if ((vec = calloc(1,sizeof(BKNodeVec))) == NULL)
371     {
372         addError(CRITICAL,"Critical lack of memory in %s at line %d",
373             __FILE__,__LINE__);
374         return NULL;
375     }
376
377     if (size == -1)
378         size = BKNODEVEC_INITSIZE;
379
380     if ((ptr = calloc(size,sizeof(BKNode))) == NULL)
381     {
382         addError(CRITICAL,"Critical lack of memory in %s at line %d",
383             __FILE__,__LINE__);
384         free(vec);
385         return NULL;
386     }
387
388     for (i=0;i<size;i++)
389     {
390         if (bkConnectVecInit(&ptr[i].inNeighb,-1)<0)

```

```

391     {
392         for (j=i-1;j>=0;j--)
393         {
394             bkConnectVecEnd(&ptr[j].inNeighb);
395             bkConnectVecEnd(&ptr[j].outNeighb);
396         }
397         addError(CRITICAL,"Unable to initialize structure in %s at line %d",
398             __FILE__,__LINE__);
399         free(vec);
400         free(ptr);
401         return NULL;
402     }
403     else if (bkConnectVecInit(&ptr[i].outNeighb,-1)<0)
404     {
405         bkConnectVecEnd(&ptr[i].inNeighb);
406         for (j=i-1;j>=0;j--)
407         {
408             bkConnectVecEnd(&ptr[j].inNeighb);
409             bkConnectVecEnd(&ptr[j].outNeighb);
410         }
411         addError(CRITICAL,"Unable to initialize structure in %s at line %d",
412             __FILE__,__LINE__);
413         free(vec);
414         free(ptr);
415         return NULL;
416     }
417 }
418
419 vec->size = size;
420 vec->top = 0;
421 vec->cont = ptr;
422
423 return vec;
424 }

```

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](#).

```

570 {
571     if (vec == NULL || vec->cont == NULL || val == NULL)
572     {
573         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
574             __FILE__,__LINE__);
575         return -1;
576     }
577
578     if (vec->top == 0)
579     {
580         addError(CRITICAL,"Pop on empty stack in %s at line %d",
581             __FILE__,__LINE__);
582         return -1;
583     }
584
585     if (bkConnectVecCopy(&val->inNeighb,&vec->cont[vec->top-1].inNeighb)<0)
586     {
587         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
588             __FILE__,__LINE__);
589         return -1;
590     }
591     if (bkConnectVecCopy(&val->outNeighb,&vec->cont[vec->top-1].outNeighb)<0)

```

```

592     {
593         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
594                 __FILE__,__LINE__);
595         return -1;
596     }
597     val->nodeId = vec->cont[vec->top-1].nodeId;
598     val->neighbInd = vec->cont[vec->top--].neighbInd;
599
600     return 0;
601 }

```

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().

```

533 {
534     if (vec == NULL || vec->cont == NULL)
535     {
536         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
537                 __FILE__,__LINE__);
538         return -1;
539     }
540
541     if (vec->top >= vec->size)
542     {
543         if (bkNodeVecResize(vec,vec->size*2)<0)
544         {
545             addError(CRITICAL,"Critical lack of memory in %s at line %d",
546                     __FILE__,__LINE__);
547             return -1;
548         }
549     }
550
551     if (bkConnectVecCopy(&vec->cont[vec->top].inNeighb,&val->inNeighb)<0)
552     {
553         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
554                 __FILE__,__LINE__);
555         return -1;
556     }
557     if (bkConnectVecCopy(&vec->cont[vec->top].outNeighb,&val->outNeighb)<0)
558     {
559         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
560                 __FILE__,__LINE__);
561         return -1;
562     }
563     vec->cont[vec->top].nodeId = val->nodeId;
564     vec->cont[vec->top++].neighbInd = val->neighbInd;
565
566     return 0;
567 }

```

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()`.

```

604 {
605     void *ptr=NULL;
606     long i;
607
608     if (vec == NULL || vec->cont == NULL)
609     {
610         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
611                 __FILE__,__LINE__);
612         return -1;
613     }
614
615     if ((ptr = (BKNode*) realloc(vec->cont, newsize*sizeof(BKNode))) == NULL)
616     {
617         addError(CRITICAL,"Critical lack of memory in %s at line %d",
618                 __FILE__,__LINE__);
619         return -1;
620     }
621
622     memset(ptr+(vec->size*sizeof(BKNode)) , 0, (newsize-vec->size)*sizeof(BKNode));
623     vec->cont = ptr;
624
625     for (i=vec->size;i<newsize;i++)
626     {
627         if (bkConnectVecInit(&((BKNode*) ptr)[i].inNeighb,-1)<0 ||
628             bkConnectVecInit(&((BKNode*) ptr)[i].outNeighb,-1)<0)
629         {
630             addError(CRITICAL,"Unable to initialize structure in %s at line %d",
631                     __FILE__,__LINE__);
632             return -1;
633         }
634     }
635     vec->size = newsize;
636
637     return 0;
638 }
```

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`.

```

667 {
668     if (vec == NULL || vec->cont == NULL)
669     {
670         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
671                 __FILE__,__LINE__);
672         return -1;
673     }
674
675     if (index < 0)
676     {
677         addError(CRITICAL,"Bad argument (index < 0) in %s at line %d",
678                 __FILE__,__LINE__);
679         return -1;
680     }

```

```

681
682     if (index >= vec->size)
683     {
684         if (bkNodeVecResize(vec,max(vec->size * 2,index+1))<0)
685         {
686             addError(CRITICAL,"Unable to resize node vector in %s at line %d",
687                     __FILE__,__LINE__);
688             return -1;
689         }
690     }
691
692     if (bkConnectVecCopy(&vec->cont[index].inNeighb,&val->inNeighb)<0)
693     {
694         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
695                 __FILE__,__LINE__);
696         return -1;
697     }
698     if (bkConnectVecCopy(&vec->cont[index].outNeighb,&val->outNeighb)<0)
699     {
700         addError(CRITICAL,"Unable to copy the list of neighbours in %s at line %d",
701                 __FILE__,__LINE__);
702         return -1;
703     }
704     vec->cont[index].nodeId = val->nodeId;
705     vec->cont[index].neighbInd = val->neighbInd;
706     vec->top=max(vec->top,index+1);
707
708     return 0;
709 }

```

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().

```

743 {
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 }

```

4.25.2.23 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().

```

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",
794             __FILE__,__LINE__);
795         longListEnd(&toDoNodes);
796         return -1;
797     }
798
799     memset(&tmpNode,0,sizeof(BKNode));
800     if (bkConnectVecInit(&tmpNode.inNeighb,-1)<0)
801     {
802         addError(CRITICAL,"Unable to initialize the temporary node in %s at line %d",
803             __FILE__,__LINE__);
804         longListEnd(&toDoNodes);
805         free(activeFlags);
806         return -1;
807     }
808     if (bkConnectVecInit(&tmpNode.outNeighb,-1)<0)
809     {
810         addError(CRITICAL,"Unable to initialize the temporary node in %s at line %d",
811             __FILE__,__LINE__);
812         longListEnd(&toDoNodes);
813         free(activeFlags);
814         bkConnectVecEnd(&tmpNode.inNeighb);
815         return -1;
816     }
817
818     if (longListPushBack(&toDoNodes,src)<0)

```

```

819 {
820     addError(CRITICAL,"Unable to push back on list of longs in %s at line %d",
821         __FILE__, __LINE__);
822     longListEnd(&toDoNodes);
823     free(activeFlags);
824     bkConnectVecEnd(&tmpNode.inNeighb);
825     bkConnectVecEnd(&tmpNode.outNeighb);
826     return -1;
827 }
828 activeFlags[src]=1;
829 while (toDoNodes.top>0)
830 {
831     if (longListPopBack(&toDoNodes,&nd)<0)
832     {
833         addError(CRITICAL,"Unable to pop back on list of longs in %s at line %d",
834             __FILE__, __LINE__);
835         longListEnd(&toDoNodes);
836         free(activeFlags);
837         bkConnectVecEnd(&tmpNode.inNeighb);
838         bkConnectVecEnd(&tmpNode.outNeighb);
839         return -1;
840     }
841     tmpNode.inNeighb.top=0;
842     if ((tmpNeighb=DBgetNodeInNeighb(dataBase,nd))==NULL)
843     {
844         addError(CRITICAL,"Unable to get the list of neighbours in %s at line %d",
845             __FILE__, __LINE__);
846         longListEnd(&toDoNodes);
847         free(activeFlags);
848         bkConnectVecEnd(&tmpNode.inNeighb);
849         bkConnectVecEnd(&tmpNode.outNeighb);
850         return -1;
851     }
852     for (i=0;i<tmpNeighb->top;i++)
853     {
854         if (activeFlags[tmpNeighb->cont[i]]==2)
855         {
856             nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
857             for (j=0;(j<nodePtr->outNeighb.top) && (nodePtr->outNeighb.cont[j].neighbId!=nd);j++);
858             if (j<nodePtr->outNeighb.top)
859             {
860                 tmpConn.neighbId=tmpNeighb->cont[i];
861                 tmpConn.linkState=nodePtr->outNeighb.cont[j].linkState;
862                 memset(&tmpConn.info,0,sizeof(BKConnectInfo));
863                 memcpy(tmpConn.info.gain,nodePtr->outNeighb.cont[j].info.gain,NB_OA*sizeof(double));
864                 if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)
865                 {
866                     addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
867                         __FILE__, __LINE__);
868                     longListEnd(&toDoNodes);
869                     free(activeFlags);
870                     bkConnectVecEnd(&tmpNode.inNeighb);
871                     bkConnectVecEnd(&tmpNode.outNeighb);
872                     return -1;
873                 }
874             }
875         }
876     }
877     else
878     {
879         tmpConn.neighbId=tmpNeighb->cont[i];
880         tmpConn.linkState=DBgetLinkState(dataBase,tmpNeighb->cont[i],nd);
881         memset(&tmpConn.info,0,sizeof(BKConnectInfo));
882         if (isValidRequestLink(dataBase,tmpNeighb->cont[i],nd,
883             tmpConn.linkState,req,tmpConn.info.gain))
884         {
885             if (bkConnectVecPushBack(&tmpNode.inNeighb,&tmpConn)<0)

```

```

886         {
887             addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
888                     __FILE__, __LINE__);
889             longListEnd(&toDoNodes);
890             free(activeFlags);
891             bkConnectVecEnd(&tmpNode.inNeighb);
892             bkConnectVecEnd(&tmpNode.outNeighb);
893             return -1;
894         }
895     }
896 }
897
898 tmpNode.outNeighb.top=0;
899 if ((tmpNeighb=DBgetNodeOutNeighb(dataBase,nd))==NULL)
900 {
901     addError(CRITICAL,"Unable to get the list of neighbours in %s at line %d",
902             __FILE__, __LINE__);
903     longListEnd(&toDoNodes);
904     free(activeFlags);
905     bkConnectVecEnd(&tmpNode.inNeighb);
906     bkConnectVecEnd(&tmpNode.outNeighb);
907     return -1;
908 }
909 for (i=0;i<tmpNeighb->top;i++)
910 {
911     if (activeFlags[tmpNeighb->cont[i]]==2)
912     {
913         nodePtr=&(topo->nodeVec.cont[topo->nodeInd.cont[tmpNeighb->cont[i]]]);
914         for (j=0;(j<nodePtr->inNeighb.top) && (nodePtr->inNeighb.cont[j].neighbId!=nd);j++);
915         if (j<nodePtr->inNeighb.top)
916         {
917             tmpConn.neighbId=tmpNeighb->cont[i];
918             tmpConn.linkState=nodePtr->inNeighb.cont[j].linkState;
919             memset(&tmpConn.info,0,sizeof(BKConnectInfo));
920             memcpy(tmpConn.info.gain,nodePtr->inNeighb.cont[j].info.gain,NB_OA*sizeof(double));
921             if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
922             {
923                 addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
924                         __FILE__, __LINE__);
925                 longListEnd(&toDoNodes);
926                 free(activeFlags);
927                 bkConnectVecEnd(&tmpNode.inNeighb);
928                 bkConnectVecEnd(&tmpNode.outNeighb);
929                 return -1;
930             }
931         }
932     }
933 }
934 else
935 {
936     tmpConn.neighbId=tmpNeighb->cont[i];
937     tmpConn.linkState=DBgetLinkState(dataBase,nd,tmpNeighb->cont[i]);
938     memset(&tmpConn.info,0,sizeof(BKConnectInfo));
939     if (isValidRequestLink(dataBase,nd,tmpNeighb->cont[i],
940                           tmpConn.linkState,req,tmpConn.info.gain))
941     {
942         if (bkConnectVecPushBack(&tmpNode.outNeighb,&tmpConn)<0)
943         {
944             addError(CRITICAL,"Unable to push back neighbour in %s at line %d",
945                     __FILE__, __LINE__);
946             longListEnd(&toDoNodes);
947             free(activeFlags);
948             bkConnectVecEnd(&tmpNode.inNeighb);
949             bkConnectVecEnd(&tmpNode.outNeighb);
950             return -1;
951         }
952     }

```

```

953     }
954     if (activeFlags[tmpNeighb->cont[i]]==0)
955     {
956         if (longListPushBack(&toDoNodes,tmpNeighb->cont[i])<0)
957         {
958             addError(CRITICAL,"Unable to push back on list of longs in %s at line %d",
959                     __FILE__,__LINE__);
960             longListEnd(&toDoNodes);
961             free(activeFlags);
962             return -1;
963         }
964         activeFlags[tmpNeighb->cont[i]]=1;
965     }
966 }
967
968 tmpNode.nodeId=nd;
969 tmpNode.neighbInd=-1;
970 if (bkNodeVecPushBack(&topo->nodeVec,&tmpNode)<0)
971 {
972     addError(CRITICAL,"Unable to push back node in %s at line %d",
973             __FILE__,__LINE__);
974     longListEnd(&toDoNodes);
975     free(activeFlags);
976     bkConnectVecEnd(&tmpNode.inNeighb);
977     bkConnectVecEnd(&tmpNode.outNeighb);
978     return -1;
979 }
980
981 if (longVecSet(&topo->nodeInd,nd,(topo->nodeVec.top-1))<0)
982 {
983     addError(CRITICAL,"Unable to set node index in %s at line %d",
984             __FILE__,__LINE__);
985     longListEnd(&toDoNodes);
986     free(activeFlags);
987     bkConnectVecEnd(&tmpNode.inNeighb);
988     bkConnectVecEnd(&tmpNode.outNeighb);
989     return -1;
990 }
991
992 activeFlags[nd]=2;
993 }
994
995 if (((topo->nbNodes=DBgetNbNodes(dataBase))<0)||
996     ((topo->nbLinks=DBgetNbLinks(dataBase))<0))
997 {
998     addError(CRITICAL,"Unable to get number of nodes and links in %s at line %d",
999             __FILE__,__LINE__);
1000     longListEnd(&toDoNodes);
1001     free(activeFlags);
1002     bkConnectVecEnd(&tmpNode.inNeighb);
1003     bkConnectVecEnd(&tmpNode.outNeighb);
1004     return -1;
1005 }
1006
1007 longListEnd(&toDoNodes);
1008 free(activeFlags);
1009 bkConnectVecEnd(&tmpNode.inNeighb);
1010 bkConnectVecEnd(&tmpNode.outNeighb);
1011
1012 return 0;
1013 }

```

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().

```

1078 {
1079     if (req==NULL)
1080     {
1081         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1082             __FILE__,__LINE__);
1083         return -1;
1084     }
1085
1086     if (req->path.top<2 || req->path.cont[0]<0 ||
1087         req->path.cont[req->path.top-1]<0)
1088     {
1089         addError(CRITICAL,"Bad requested path format in %s at line %d",
1090             __FILE__,__LINE__);
1091         return -1;
1092     }
1093
1094     return req->path.cont[req->path.top-1];
1095 }
```

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().

```

1058 {
1059     if (req==NULL)
1060     {
1061         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1062             __FILE__,__LINE__);
1063         return -1;
1064     }
1065
1066     if (req->path.top<2 || req->path.cont[0]<0 ||
1067         req->path.cont[req->path.top-1]<0)
1068     {
1069         addError(CRITICAL,"Bad requested path format in %s at line %d",
1070             __FILE__,__LINE__);
1071         return -1;
1072     }
1073
1074     return req->path.cont[0];
1075 }
```

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().

```

1403 {
1404     bool process=FALSE;
1405     long i,j,k,l,top;
1406     double tmpSum;
1407
1408     if (topo == NULL)
1409     {
1410         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1411             __FILE__,__LINE__);
1412         return -1;
1413     }
1414
1415     for (i=0;i<NB_OA;i++)
1416     {
1417         if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
1418         {
1419             process=TRUE;
1420         }
1421     }
1422
1423     if (process)
1424     {
1425         for (k=0;k<NB_OA;k++)
1426         {
1427             topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]=0;
1428         }
1429         for (i=0;i<topo->nodeVec.top;i++)
1430         {
1431             top=topo->nodeVec.cont[i].inNeighb.top;
1432             for (j=0;j<top;j++)
1433             {
1434                 for (k=0;k<NB_OA;k++)
1435                 {
1436                     tmpSum=0;
1437                     for (l=0;l<NB_PREEMPTION;l++)
1438                     {
1439                         tmpSum+=topo->nodeVec.cont[i].inNeighb.cont[j].linkState->pbw[k][l];
1440                     }
1441                     topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[k]+=tmpSum/
1442                     topo->nodeVec.cont[i].inNeighb.cont[j].linkState->cap[k];
1443                 }
1444             }
1445         }
1446     }
1447
1448     return 0;
1449 }

```

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().

```

716 {
717     if (topo == NULL)
718     {
719         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
720             __FILE__,__LINE__);
721         return -1;
722     }

```

```

723
724     if (bkNodeVecInit(&topo->nodeVec,-1)<0)
725     {
726         addError(CRITICAL,"Unable to initialize node vector in %s at line %d",
727             __FILE__,__LINE__);
728         return -1;
729     }
730
731     if (longVecInit(&topo->nodeInd,size)<0)
732     {
733         addError(CRITICAL,"Unable to initialize long vector in %s at line %d",
734             __FILE__,__LINE__);
735         bkNodeVecEnd(&topo->nodeVec);
736         return -1;
737     }
738
739     return 0;
740 }

```

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().

```

1453 {
1454     double score=0,totBW[NB_OA],newSum,rerouteScore=0;
1455     long i,j;
1456
1457     if (topo == NULL || connect == NULL)
1458     {
1459         addError(CRITICAL,"Bad argument (NULL) in %s at line %d",
1460             __FILE__,__LINE__);
1461         return HUGE_VAL;
1462     }
1463
1464     score=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.cost;
1465
1466     for (i=0;i<NB_OA;i++)
1467     {
1468         totBW[i]=0;
1469         for (j=0;j<NB_PREEMPTION;j++)
1470         {
1471             totBW[i]+=connect->linkState->pbw[i][j];
1472         }
1473
1474         if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
1475         {
1476             score+=damoteConfig.primaryComputationConfig.loadBal[i]
1477                 *((totBW[i]+req->bw[i])/connect->linkState->cap[i])
1478                 *((totBW[i]+req->bw[i])/connect->linkState->cap[i]);
1479             score+=damoteConfig.primaryComputationConfig.loadBal[i]
1480                 *(totBW[i]/connect->linkState->cap[i])
1481                 *(totBW[i]/connect->linkState->cap[i]);
1482         }
1483     }
1484 }

```

```

1483         newSum=topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkS
1484         if (__isinf(newSum))
1485         {
1486             return HUGE_VAL;
1487         }
1488
1489         score+=damoteConfig.primaryComputationConfig.loadBal[i]
1490             *(-1/(double)topo->nbLinks)*newSum*newSum;
1491         score+=damoteConfig.primaryComputationConfig.loadBal[i]
1492             *(1/(double)topo->nbLinks)*topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]
1493             *topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i];
1494     }
1495     if (damoteConfig.primaryComputationConfig.load[i]!=0)
1496     {
1497         score+=damoteConfig.primaryComputationConfig.load[i]*req->bw[i];
1498     }
1499     if (damoteConfig.primaryComputationConfig.sqload[i]!=0)
1500     {
1501         score+=damoteConfig.primaryComputationConfig.sqload[i]
1502             *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i]);
1503     }
1504     if (damoteConfig.primaryComputationConfig.relLoad[i]!=0)
1505     {
1506         score+=damoteConfig.primaryComputationConfig.relLoad[i]
1507             *req->bw[i]/connect->linkState->cap[i];
1508     }
1509     if (damoteConfig.primaryComputationConfig.sqRelLoad[i]!=0)
1510     {
1511         score+=damoteConfig.primaryComputationConfig.sqRelLoad[i]
1512             *(req->bw[i]*req->bw[i]+2*req->bw[i]*totBW[i])
1513             /(connect->linkState->cap[i]*connect->linkState->cap[i]);
1514     }
1515     if (damoteConfig.primaryComputationConfig.delay[i]!=0)
1516     {
1517         score+=damoteConfig.primaryComputationConfig.delay[i]
1518             *((1/(connect->linkState->cap[i]-totBW[i]-req->bw[i]))
1519             -(1/(connect->linkState->cap[i]-totBW[i])));
1520     }
1521 }
1522
1523 for (i=0;i<NB_OA;i++)
1524 {
1525     if (damoteConfig.primaryComputationConfig.rerouting[i]!=0)
1526     {
1527         rerouteScore+=damoteConfig.primaryComputationConfig.rerouting[i]*
1528             makeRerouteScore(req,connect->info.gain,connect->linkState,i);
1529     }
1530 }
1531
1532 score+=rerouteScore*(score>0?1:0)*score;
1533
1534 return score;
1535 }

```

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::in-Neighb, BKNode::neighbInd, BKNodeInfo::newNeighbInd, BKNode::nodeId, BKTopology::nodeInd, and BKTopology::nodeVec.

Referenced by bellmanKalaba().

```
1587 {
```



```

1588     BKNode* tmpNode;
1589
1590
1591     tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[src]];
1592     while (tmpNode->neighbInd!=-1 && tmpNode->nodeId!=dst)
1593     {
1594         tmpNode=&topo->nodeVec.cont[topo->nodeInd.cont[tmpNode->inNeighb.cont[tmpNode->info.newNeighb
1595     ]
1596
1597     if (tmpNode->nodeId==dst)
1598         return 0;
1599
1600     return 1;
1601 }

```

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`.

```

1016 {
1017     BKNode *tmpNode;
1018     long i,j;
1019
1020     for (i=0;i<topo->nodeInd.top;i++)
1021     {
1022         tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[i]);
1023         if (tmpNode!=NULL)
1024         {
1025             if (i!=tmpNode->nodeId)
1026             {
1027                 addError(CRITICAL,"Topology unconsistency in %s at line %d",
1028                     __FILE__,__LINE__);
1029                 return -1;
1030             }
1031
1032             printf("Node %ld\n-----\n",i);
1033             printf("Incoming neighbors : \n");
1034
1035             for (j=0; j<tmpNode->inNeighb.top; j++)
1036             {
1037                 printf("%ld ", tmpNode->inNeighb.cont[j].neighbId);
1038             }
1039
1040             printf("\nOutgoing neighbors : \n");
1041
1042             for (j=0; j<tmpNode->outNeighb.top; j++)
1043             {
1044                 printf("%ld ", tmpNode->outNeighb.cont[j].neighbId);
1045             }
1046             printf("\n");
1047
1048             printf("Chosen Neighbour Index: %ld \n",tmpNode->neighbInd);
1049
1050             printf("\n");
1051         }
1052     }
1053
1054     return 0;
1055 }

```

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().

```

1539 {
1540     long i;
1541
1542     if (topo == NULL || connect == NULL)
1543     {
1544         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
1545                 __FILE__, __LINE__);
1546         return -1;
1547     }
1548
1549     for (i=0; i<NB_OA; i++)
1550     {
1551         if (damoteConfig.primaryComputationConfig.loadBal[i]!=0)
1552         {
1553             topo->nodeVec.cont[topo->nodeInd.cont[dst]].info.newSum[i]=
1554                 topo->nodeVec.cont[topo->nodeInd.cont[src]].info.sum[i]+(req->bw[i]/connect->linkStat
1555         }
1556     }
1557
1558     return 0;
1559 }
```

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().

```

1098 {
1099     BKNode *tmpNode;
1100     long i, src, dst, nd;
1101
1102     if (topo == NULL || req==NULL)
1103     {
1104         addError(CRITICAL, "Bad argument (NULL) in %s at line %d",
1105                 __FILE__, __LINE__);
1106         return -1;
1107     }
1108
1109     if ((src=getRequestSrc(req))<0)
1110     {
1111         addError(CRITICAL, "Unable to get requested source in %s at line %d",
1112                 __FILE__, __LINE__);
1113         return -1;
1114     }
1115
1116     if ((dst=getRequestDst(req))<0)
1117     {
1118         addError(CRITICAL, "Unable to get requested source in %s at line %d",
```

```
1119         __FILE__, __LINE__);
1120     return -1;
1121 }
1122
1123 req->path.top=0;
1124 nd=dst;
1125 if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
1126 {
1127     addError(CRITICAL,"Undetermined error in %s at line %d",
1128         __FILE__, __LINE__);
1129     return -1;
1130 }
1131 while (nd!=src)
1132 {
1133     if (tmpNode->neighbInd < 0)
1134     {
1135         addError(CRITICAL,"Destination unreachable in %s at line %d",
1136             __FILE__, __LINE__);
1137         return -1;
1138     }
1139     if (longListPushBack(&req->path,nd)<0)
1140     {
1141         addError(CRITICAL,"Undetermined error in %s at line %d",
1142             __FILE__, __LINE__);
1143         return -1;
1144     }
1145     nd=tmpNode->inNeighb.cont[tmpNode->neighbInd].neighbId;
1146     if ((tmpNode=bkNodeVecGet(&topo->nodeVec,topo->nodeInd.cont[nd]))==NULL)
1147     {
1148         addError(CRITICAL,"Undetermined error in %s at line %d",
1149             __FILE__, __LINE__);
1150         return -1;
1151     }
1152 }
1153 if (longListPushBack(&req->path,nd)<0)
1154 {
1155     addError(CRITICAL,"Undetermined error in %s at line %d",
1156         __FILE__, __LINE__);
1157     return -1;
1158 }
1159
1160 for (i=0;i<req->path.top/2;i++)
1161 {
1162     nd=req->path.cont[i];
1163     req->path.cont[i]=req->path.cont[req->path.top-1-i];
1164     req->path.cont[req->path.top-1-i]=nd;
1165 }
1166
1167 return 0;
1168 }
```

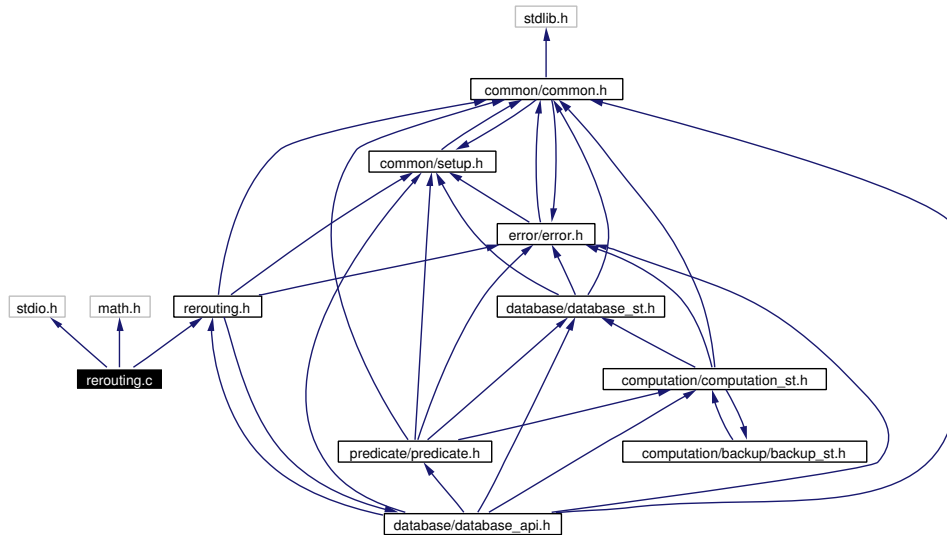
4.26 rerouting.c File Reference

```
#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\(\)](#).

```

8 {
9     int i,j,p;
10    double need;
11
12    if (idList==NULL || idList->cont==NULL || state==NULL ||
13        lspList==NULL || lspList->cont==NULL)
14    {

```

```

15     addError(CRITICAL,"Invalid arguments in %s at line %d",
16               __FILE__,__LINE__);
17     return -1;
18 }
19
20 for (i=1;i<NB_OA;i++)
21 {
22     if (toGain[i]>DIGIT_PRECISION)
23     {
24         addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
25                 __FILE__,__LINE__);
26         return -1;
27     }
28     for (j=0;j<NB_PREEMPTION;j++)
29     {
30         if (state->rbw[i][j]>DIGIT_PRECISION)
31         {
32             addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
33                     __FILE__,__LINE__);
34             return -1;
35         }
36     }
37 }
38
39 need=toGain[0];
40 j=0;
41 while (need>DIGIT_PRECISION && j<lspList->top)
42 {
43     p=lspList->cont[j]->precedence;
44     if (p<=precedence)
45     {
46         addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",
47                 __FILE__,__LINE__);
48         return -1;
49     }
50
51     if (need >= state->rbw[0][p])
52     {
53         while (j<lspList->top && lspList->cont[j]->precedence==p)
54         {
55             longListPushBack(idList,lspList->cont[j]->id);
56             j++;
57         }
58         need=need-state->rbw[0][p];
59         p=p-1;
60     }
61     else while (need>DIGIT_PRECISION && j<lspList->top)
62     {
63         if (lspList->cont[j]->bw[0]<=need)
64         {
65             longListPushBack(idList,lspList->cont[j]->id);
66             need=need-lspList->cont[j]->bw[0];
67             j++;
68         }
69         else
70         {
71             while (j<lspList->top && lspList->cont[j]->bw[0]>=need && lspList->cont[j]->precedence=
72                     j++;
73             longListPushBack(idList,lspList->cont[j-1]->id);
74             need=need-lspList->cont[j-1]->bw[0];
75         }
76     }
77 }
78
79 if (need>DIGIT_PRECISION)
80 {
81     addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",

```

```

82         __FILE__, __LINE__);
83     return -1;
84 }
85
86 return 0;
87 }

```

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().

```

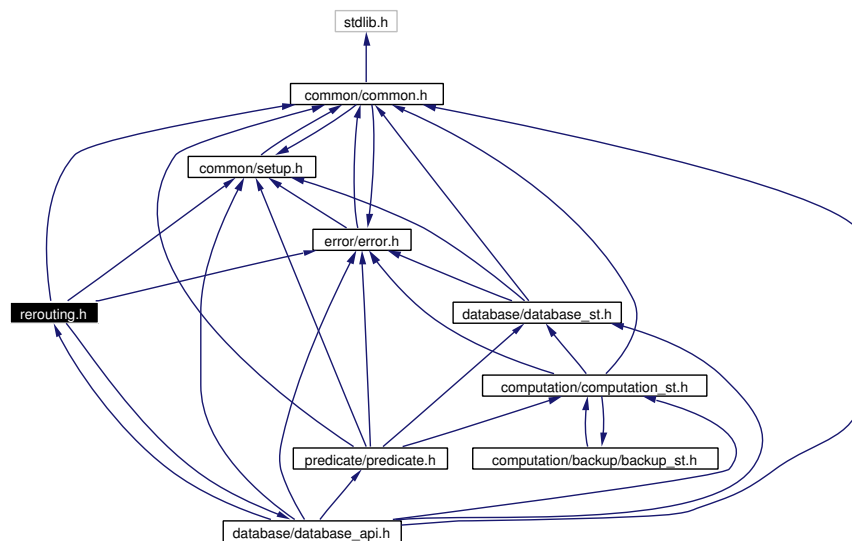
91 {
92     int curPrec;
93     double bwGained, score=0;
94
95     bwGained=0;
96     curPrec=NB_PREEMPTION-1;
97     while (bwGained<gain[oa])
98     {
99         if (curPrec<=req->precedence)
100         {
101             addError(CRITICAL, "internal error: impossible to realize gain in %s at line %d",
102                     __FILE__, __LINE__);
103             return HUGE_VAL;
104         }
105         score=score+damoteConfig.reroutingConfig.scoreCoef[oa][curPrec]*
106             min(ls->rbw[oa][curPrec], gain[oa]-bwGained);
107         bwGained=bwGained+
108             min(ls->rbw[oa][curPrec], gain[oa]-bwGained);
109         curPrec--;
110     }
111
112     return score/req->bw[oa];
113 }

```

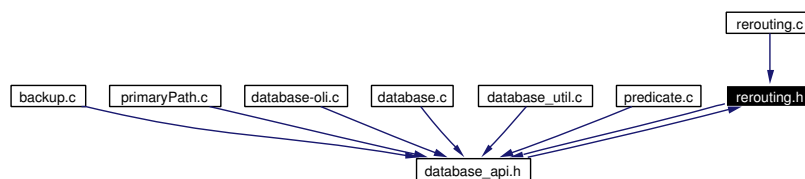
4.27 rerouting.h File Reference

```
#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().

```

8 {
9     int i,j,p;
10    double need;
11
12    if (idList==NULL || idList->cont==NULL || state==NULL ||
13        lspList==NULL || lspList->cont==NULL)
14    {
15        addError(CRITICAL,"Invalid arguments in %s at line %d",
16                __FILE__,__LINE__);
17        return -1;
18    }
19
20    for (i=1;i<NB_OA;i++)
21    {
22        if (toGain[i]>DIGIT_PRECISION)
23        {
24            addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
25                    __FILE__,__LINE__);
26            return -1;
27        }
28        for (j=0;j<NB_PREEMPTION;j++)
29        {
30            if (state->rbw[i][j]>DIGIT_PRECISION)
31            {
32                addError(CRITICAL,"Only one ordered aggregate taken into account in %s at line %d",
33                        __FILE__,__LINE__);
34                return -1;
35            }
36        }
37    }
38
39    need=toGain[0];
40    j=0;
41    while (need>DIGIT_PRECISION && j<lspList->top)
42    {
43        p=lspList->cont[j]->precedence;
44        if (p<=precedence)
45        {
46            addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",
47                    __FILE__,__LINE__);
48            return -1;
49        }
50
51        if (need >= state->rbw[0][p])
52        {
53            while (j<lspList->top && lspList->cont[j]->precedence==p)
54            {
55                longListPushBack(idList,lspList->cont[j]->id);
56                j++;
57            }
58            need=need-state->rbw[0][p];
59            p=p-1;
60        }
61        else while (need>DIGIT_PRECISION && j<lspList->top)
62        {
63            if (lspList->cont[j]->bw[0]<=need)
64            {
65                longListPushBack(idList,lspList->cont[j]->id);
66                need=need-lspList->cont[j]->bw[0];
67                j++;
68            }

```



```

69         else
70         {
71             while (j<lspList->top && lspList->cont[j]->bw[0]>=need && lspList->cont[j]->precedence=
72                 j++;
73             longListPushBack(idList,lspList->cont[j-1]->id);
74             need=need-lspList->cont[j-1]->bw[0];
75         }
76     }
77 }
78
79 if (need>DIGIT_PRECISION)
80 {
81     addError(CRITICAL,"Not enough preemptable bandwidth for LSP selection in %s at line %d",
82         __FILE__,__LINE__);
83     return -1;
84 }
85
86 return 0;
87 }

```

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()`.

```

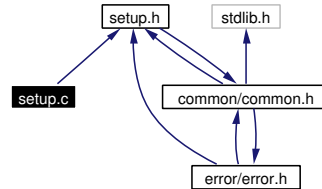
91 {
92     int curPrec;
93     double bwGained,score=0;
94
95     bwGained=0;
96     curPrec=Nb_PREEMPTION-1;
97     while (bwGained<gain[oa])
98     {
99         if (curPrec<=req->precedence)
100         {
101             addError(CRITICAL,"internal error: impossible to realize gain in %s at line %d",
102                 __FILE__,__LINE__);
103             return HUGE_VAL;
104         }
105         score=score+damoteConfig.reroutingConfig.scoreCoef[oa][curPrec]*
106             min(ls->rbw[oa][curPrec],gain[oa]-bwGained);
107         bwGained=bwGained+
108             min(ls->rbw[oa][curPrec],gain[oa]-bwGained);
109         curPrec--;
110     }
111
112     return score/req->bw[oa];
113 }

```

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](#)

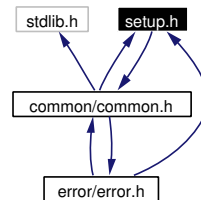
Definition at line 3 of file setup.c.

Referenced by activateNodeInfo(), capacityClause(), initScore(), isValidRequestLink(), makeRerouteScore(), makeScore(), and updateNodeInfoOnElect().

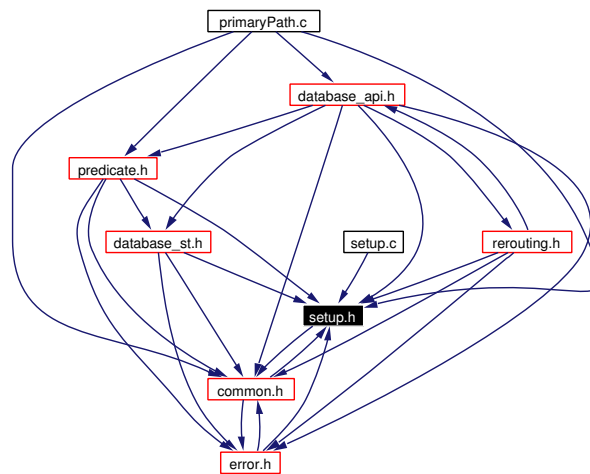
4.29 setup.h File Reference

```
#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

- activateNodeInfo
 - primaryPath.c, [360](#)
 - primaryPath_util.h, [394](#)
- addError
 - error.c, [341](#)
 - error.h, [346](#)
- allocatedMemory
 - common.c, [134](#)
 - common.h, [154](#)
- allowReroute
 - PredicateConfig-, [56](#)
- ANDERROR
 - common.h, [137](#)
- avl.c, [61](#)
 - avl_allocator_default, [75](#)
 - avl_assert_delete, [76](#)
 - avl_assert_insert, [76](#)
 - avl_copy, [62](#)
 - avl_create, [63](#)
 - avl_delete, [64](#)
 - avl_destroy, [67](#)
 - avl_find, [67](#)
 - avl_free, [67](#)
 - avl_insert, [68](#)
 - avl_malloc, [68](#)
 - avl_probe, [68](#)
 - avl_replace, [70](#)
 - avl_t_copy, [70](#)
 - avl_t_cur, [71](#)
 - avl_t_find, [71](#)
 - avl_t_first, [71](#)
 - avl_t_init, [72](#)
 - avl_t_insert, [72](#)
 - avl_t_last, [73](#)
 - avl_t_next, [73](#)
 - avl_t_prev, [74](#)
 - avl_t_replace, [75](#)
- avl.h, [77](#)
 - avl_allocator_default, [92](#)
 - avl_assert_delete, [79](#)
 - avl_assert_insert, [79](#)
 - avl_comparison_func, [78](#)
 - avl_copy, [79](#)
 - avl_copy_func, [78](#)
 - avl_count, [78](#)
 - avl_create, [80](#)
 - avl_delete, [81](#)
 - avl_destroy, [83](#)
 - avl_find, [84](#)
 - avl_free, [84](#)
 - AVL_H, [78](#)
 - avl_insert, [85](#)
 - avl_item_func, [79](#)
 - avl_malloc, [85](#)
 - AVL_MAX_HEIGHT, [78](#)
 - avl_probe, [85](#)
 - avl_replace, [87](#)
 - avl_t_copy, [87](#)
 - avl_t_cur, [87](#)
 - avl_t_find, [88](#)
 - avl_t_first, [88](#)
 - avl_t_init, [89](#)
 - avl_t_insert, [89](#)
 - avl_t_last, [90](#)
 - avl_t_next, [90](#)
 - avl_t_prev, [91](#)
 - avl_t_replace, [92](#)
- avl_alloc
 - avl_table, [7](#)
- avl_allocator_default
 - avl.c, [75](#)
 - avl.h, [92](#)
- avl_assert_delete
 - avl.c, [76](#)
 - avl.h, [79](#)
- avl_assert_insert
 - avl.c, [76](#)
 - avl.h, [79](#)
- avl_balance
 - avl_node, [5](#)
- avl_compare
 - avl_table, [7](#)
- avl_comparison_func
 - avl.h, [78](#)
- avl_copy
 - avl.c, [62](#)
 - avl.h, [79](#)
- avl_copy_func
 - avl.h, [78](#)
- avl_count

- avl.h, 78
 - avl_table, 7
- avl_create
 - avl.c, 63
 - avl.h, 80
- avl_data
 - avl_node, 5
- avl_delete
 - avl.c, 64
 - avl.h, 81
- avl_destroy
 - avl.c, 67
 - avl.h, 83
- avl_find
 - avl.c, 67
 - avl.h, 84
- avl_free
 - avl.c, 67
 - avl.h, 84
- avl_generation
 - avl_table, 7
 - avl_traverser, 9
- AVL_H
 - avl.h, 78
- avl_height
 - avl_traverser, 9
- avl_insert
 - avl.c, 68
 - avl.h, 85
- avl_item_func
 - avl.h, 79
- avl_link
 - avl_node, 5
- avl_malloc
 - avl.c, 68
 - avl.h, 85
- AVL_MAX_HEIGHT
 - avl.h, 78
- avl_node, 5
 - avl_balance, 5
 - avl_data, 5
 - avl_link, 5
 - avl_traverser, 9
- avl_param
 - avl_table, 7
- avl_probe
 - avl.c, 68
 - avl.h, 85
- avl_replace
 - avl.c, 70
 - avl.h, 87
- avl_root
 - avl_table, 8
- avl_stack
 - avl_traverser, 9
- avl_t_copy
 - avl.c, 70
 - avl.h, 87
- avl_t_cur
 - avl.c, 71
 - avl.h, 87
- avl_t_find
 - avl.c, 71
 - avl.h, 88
- avl_t_first
 - avl.c, 71
 - avl.h, 88
- avl_t_init
 - avl.c, 72
 - avl.h, 89
- avl_t_insert
 - avl.c, 72
 - avl.h, 89
- avl_t_last
 - avl.c, 73
 - avl.h, 90
- avl_t_next
 - avl.c, 73
 - avl.h, 90
- avl_t_prev
 - avl.c, 74
 - avl.h, 91
- avl_t_replace
 - avl.c, 75
 - avl.h, 92
- avl_table, 7
 - avl_alloc, 7
 - avl_compare, 7
 - avl_count, 7
 - avl_generation, 7
 - avl_param, 7
 - avl_root, 8
 - avl_traverser, 9
- avl_traverser, 9
 - avl_generation, 9
 - avl_height, 9
 - avl_node, 9
 - avl_stack, 9
 - avl_table, 9
- backLSPIDs
 - DBLabelSwitchedPath_, 33
- backup.c, 93
 - computeBackup, 93
 - computeCost, 103
- backup_api.h, 106
 - computeBackup, 106
- backup_st.h

- GLOBAL, 118
- LOCAL, 118
- NONE, 118
- backup_st.h, 117
 - BackupType, 117
- BackupType
 - backup_st.h, 117
- bbw
 - DBLinkState_, 38
- bellmanKalaba
 - primaryPath.c, 360
 - primaryPath_util.h, 394
- BKConnect
 - primaryPath_util.h, 393
- BKConnect_, 11
 - info, 11
 - linkState, 11
 - neighbId, 11
- BKConnectInfo
 - primaryPath_util.h, 393
- BKConnectInfo_, 12
- BKConnectInfo_
 - gain, 12
- BKConnectVec
 - primaryPath_util.h, 394
- BKConnectVec_, 13
- BKConnectVec_
 - cont, 13
 - size, 13
 - top, 13
- BKCONNECTVEC_INITSIZE
 - setup.h, 430
- bkConnectVecCopy
 - primaryPath.c, 364
 - primaryPath_util.h, 398
- bkConnectVecDestroy
 - primaryPath.c, 365
 - primaryPath_util.h, 399
- bkConnectVecEnd
 - primaryPath.c, 365
 - primaryPath_util.h, 399
- bkConnectVecGet
 - primaryPath.c, 366
 - primaryPath_util.h, 399
- bkConnectVecInit
 - primaryPath.c, 366
 - primaryPath_util.h, 400
- bkConnectVecNew
 - primaryPath.c, 367
 - primaryPath_util.h, 401
- bkConnectVecPopBack
 - primaryPath.c, 367
 - primaryPath_util.h, 401
- bkConnectVecPushBack
 - primaryPath.c, 368
 - primaryPath_util.h, 402
- bkConnectVecResize
 - primaryPath.c, 368
 - primaryPath_util.h, 402
- bkConnectVecSet
 - primaryPath.c, 369
 - primaryPath_util.h, 403
- BKNode
 - primaryPath_util.h, 394
- BKNode_, 15
 - info, 15
 - inNeighb, 15
 - neighbInd, 15
 - nodeId, 16
 - outNeighb, 16
- BKNodeInfo
 - primaryPath_util.h, 394
- BKNodeInfo_, 17
- BKNodeInfo_
 - cost, 17
 - newCost, 17
 - newNeighbInd, 17
 - newSum, 17
 - sum, 17
- BKNodeVec
 - primaryPath_util.h, 394
- BKNodeVec_, 18
- BKNodeVec_
 - cont, 18
 - size, 18
 - top, 18
- BKNODEVEC_INITSIZE
 - setup.h, 430
- bkNodeVecDestroy
 - primaryPath.c, 369
 - primaryPath_util.h, 403
- bkNodeVecEnd
 - primaryPath.c, 370
 - primaryPath_util.h, 404
- bkNodeVecGet
 - primaryPath.c, 370
 - primaryPath_util.h, 404
- bkNodeVecInit
 - primaryPath.c, 371
 - primaryPath_util.h, 405
- bkNodeVecNew
 - primaryPath.c, 372
 - primaryPath_util.h, 406
- bkNodeVecPopBack
 - primaryPath.c, 373
 - primaryPath_util.h, 407
- bkNodeVecPushBack
 - primaryPath.c, 374

- primaryPath_util.h, 408
- bkNodeVecResize
 - primaryPath.c, 374
 - primaryPath_util.h, 408
- bkNodeVecSet
 - primaryPath.c, 375
 - primaryPath_util.h, 409
- BKTopology
 - primaryPath_util.h, 394
- BKTopology_., 20
 - nbLinks, 20
 - nbNodes, 20
 - nodeInd, 20
 - nodeVec, 21
- bool
 - common.h, 139
- bw
 - DBLabelSwitchedPath_., 33
 - LSPRequest_., 51
- calloc
 - common.h, 137
- cap
 - DBLinkState_., 38
- capacityClause
 - predicate.c, 350
 - PredicateConfig_., 56
- chooseReroutedLSPs
 - rerouting.c, 422
 - rerouting.h, 425
- color
 - DBLinkState_., 38
- colorClause
 - predicate.c, 351
 - PredicateConfig_., 56
- common.c, 119
 - allocatedMemory, 134
 - dblVecCopy, 120
 - dblVecDestroy, 120
 - dblVecEnd, 121
 - dblVecGet, 121
 - dblVecInit, 121
 - dblVecNew, 122
 - dblVecPopBack, 123
 - dblVecPushBack, 123
 - dblVecResize, 124
 - dblVecSet, 124
 - longCompare, 125
 - longListInsert, 125
 - longListMerge, 126
 - longListRemove, 127
 - longListSort, 127
 - longVecCopy, 128
 - longVecDestroy, 128
 - longVecEnd, 129
 - longVecGet, 129
 - longVecInit, 130
 - longVecNew, 130
 - longVecPopBack, 131
 - longVecPushBack, 131
 - longVecResize, 132
 - longVecSet, 132
 - mycalloc, 133
 - myfree, 133
 - mymalloc, 134
 - myrealloc, 134
- common.h, 135
 - allocatedMemory, 154
 - ANDERROR, 137
 - bool, 139
 - calloc, 137
 - dblVecCopy, 140
 - dblVecDestroy, 140
 - dblVecEnd, 141
 - dblVecGet, 141
 - dblVecInit, 141
 - dblVecNew, 142
 - dblVecPopBack, 143
 - dblVecPushBack, 143
 - dblVecResize, 144
 - dblVecSet, 144
 - DoubleVec, 139
 - FALSE, 139
 - free, 137
 - LongList, 139
 - longListCopy, 137
 - longListDestroy, 138
 - longListEnd, 138
 - longListInit, 138
 - longListInsert, 145
 - longListMerge, 145
 - longListNew, 138
 - longListPopBack, 138
 - longListPushBack, 138
 - longListRemove, 147
 - longListResize, 138
 - longListSort, 147
 - LongVec, 139
 - longVecCopy, 148
 - longVecDestroy, 148
 - longVecEnd, 148
 - longVecGet, 149
 - longVecInit, 149
 - longVecNew, 150
 - longVecPopBack, 151
 - longVecPushBack, 151
 - longVecResize, 152
 - longVecSet, 152

- malloc, 138
- max, 139
- min, 139
- mycalloc, 153
- myfree, 153
- mymalloc, 153
- myrealloc, 154
- realloc, 139
- TRUE, 139
- computation.c, 155
 - lspRequestCopy, 155
 - lspRequestDestroy, 156
 - lspRequestEnd, 156
 - lspRequestInit, 157
 - lspRequestListEnd, 157
 - lspRequestListGet, 158
 - lspRequestListInit, 158
 - lspRequestListResize, 159
 - lspRequestListSet, 160
 - lspRequestListSize, 160
 - lspRequestNew, 160
- computation_api.h, 162
 - lspRequestCopy, 163
 - lspRequestDestroy, 163
 - lspRequestEnd, 164
 - lspRequestInit, 164
 - lspRequestListEnd, 165
 - lspRequestListGet, 165
 - lspRequestListInit, 165
 - lspRequestListResize, 166
 - lspRequestListSet, 167
 - lspRequestListSize, 167
 - lspRequestNew, 167
- computation_st.h, 169
 - LSPRequest, 170
 - LSPRequestList, 170
 - LSPrerouteInfo, 170
- computeBackup
 - backup.c, 93
 - backup_api.h, 106
- computeCost
 - backup.c, 103
- computePrimaryPath
 - primaryPath.c, 376
 - primaryPath_api.h, 390
- computeRBW
 - database-oli.c, 173
 - database.c, 214
- cont
 - BKConnectVec_, 13
 - BKNodeVec_, 18
 - DBLinkTab_, 40
 - DBLSPList_, 41
 - DBLSPVec_, 42
 - DBNodeVec_, 44
 - DoubleVec_, 45
 - LongVec_, 49
 - LSPRequestList_, 53
- CONTAINER_TEST
 - setup.h, 430
- cost
 - BKNodeInfo_, 17
- CPdestroyPQ
 - dijkstra.c, 329
 - dijkstra.h, 336
- CPdestroyTN
 - dijkstra.c, 330
 - dijkstra.h, 336
- CPDijkNode
 - dijkstra.h, 336
- CPDijkNode_, 22
- CPDijkNode_
 - from, 22
 - marked, 22
 - node, 22
 - val, 22
- CPendPQ
 - dijkstra.c, 330
 - dijkstra.h, 337
- CPinitPQ
 - dijkstra.c, 330
 - dijkstra.h, 337
- CPinsertPQ
 - dijkstra.c, 331
 - dijkstra.h, 337
- CPnewPQ
 - dijkstra.c, 332
 - dijkstra.h, 339
- CPnewTN
 - dijkstra.c, 332
- CPpopTop
 - dijkstra.c, 333
 - dijkstra.h, 339
- CPPrioQueue
 - dijkstra.h, 336
- CPPrioQueue_, 24
- CPPrioQueue_
 - root, 24
 - size, 24
 - top, 24
- CPTreeNode
 - dijkstra.h, 336
- CPTreeNode_, 26
- CPTreeNode_
 - father, 26
 - gt, 26
 - key, 26
 - leq, 26

- node, [27](#)
- CRITICAL
 - error.h, [346](#)
- DAMOTEConfig
 - setup.h, [432](#)
- damoteConfig
 - setup.c, [428](#)
 - setup.h, [432](#)
- DAMOTEConfig_, [28](#)
 - predicateConfig, [28](#)
 - primaryComputationConfig, [28](#)
 - reroutingConfig, [28](#)
- DataBase
 - database_st.h, [285](#)
- database-oli.c, [171](#)
 - computeRBW, [173](#)
 - DBaddLink, [175](#)
 - DBaddLSP, [176](#)
 - DBaddNode, [181](#)
 - DBdestroy, [182](#)
 - DBevalLSONRemove, [182](#)
 - DBevalLSONSetup, [182](#)
 - DBgetID, [182](#)
 - DBgetLinkDst, [183](#)
 - DBgetLinkID, [183](#)
 - DBgetLinkLSPs, [183](#)
 - DBgetLinkSrc, [184](#)
 - DBgetLinkState, [184](#)
 - DBgetLSP, [185](#)
 - DBgetMaxNodeID, [185](#)
 - DBgetNbLinks, [185](#)
 - DBgetNbNodes, [185](#)
 - DBgetNodeInNeighb, [186](#)
 - DBgetNodeOutNeighb, [186](#)
 - DBlinkStateCopy, [187](#)
 - DBlinkStateDestroy, [187](#)
 - DBlinkStateEnd, [188](#)
 - DBlinkStateInit, [188](#)
 - DBlinkStateNew, [189](#)
 - DBlspCompare, [191](#)
 - DBlspCopy, [191](#)
 - DBlspDestroy, [192](#)
 - DBlspEnd, [192](#)
 - DBlspInit, [192](#)
 - DBlspListDestroy, [193](#)
 - DBlspListEnd, [194](#)
 - DBlspListInit, [194](#)
 - DBlspListInsert, [194](#)
 - DBlspListNew, [196](#)
 - DBlspListRemove, [196](#)
 - DBlspNew, [197](#)
 - DBnew, [198](#)
 - DBprintDB, [199](#)
 - DBremoveLink, [200](#)
 - DBremoveLSP, [201](#)
 - DBremoveNode, [202](#)
 - DBsetLinkState, [203](#)
 - DBupdateLSONRemove, [204](#)
 - DBupdateLSONSetup, [204](#)
 - evalLS, [204](#)
 - operation, [172](#)
 - operation_, [172](#)
 - REMOVE, [172](#)
 - SETUP, [172](#)
 - updateLS, [206](#)
- database.c, [212](#)
 - computeRBW, [214](#)
 - DBaddLink, [216](#)
 - DBaddLSP, [217](#)
 - DBaddNode, [222](#)
 - DBdestroy, [223](#)
 - DBevalLSONRemove, [224](#)
 - DBevalLSONSetup, [224](#)
 - DBgetID, [224](#)
 - DBgetLinkDst, [224](#)
 - DBgetLinkID, [225](#)
 - DBgetLinkLSPs, [225](#)
 - DBgetLinkSrc, [226](#)
 - DBgetLinkState, [226](#)
 - DBgetLSP, [227](#)
 - DBgetMaxNodeID, [227](#)
 - DBgetNbLinks, [227](#)
 - DBgetNbNodes, [228](#)
 - DBgetNodeInNeighb, [228](#)
 - DBgetNodeOutNeighb, [228](#)
 - DBlinkStateCopy, [229](#)
 - DBlinkStateDestroy, [229](#)
 - DBlinkStateEnd, [230](#)
 - DBlinkStateInit, [230](#)
 - DBlinkStateNew, [232](#)
 - DBlspCompare, [233](#)
 - DBlspCopy, [234](#)
 - DBlspDestroy, [234](#)
 - DBlspEnd, [235](#)
 - DBlspInit, [235](#)
 - DBlspListDestroy, [236](#)
 - DBlspListEnd, [237](#)
 - DBlspListInit, [237](#)
 - DBlspListInsert, [237](#)
 - DBlspListNew, [239](#)
 - DBlspListRemove, [239](#)
 - DBlspNew, [240](#)
 - DBnew, [241](#)
 - DBprintDB, [242](#)
 - DBremoveLink, [243](#)
 - DBremoveLSP, [244](#)
 - DBremoveNode, [246](#)

- DBsetLinkState, 247
- DBupdateLSOnRemove, 247
- DBupdateLSOnSetup, 247
- evalLS, 248
- operation, 213
- operation_, 213
- REMOVE, 213
- SETUP, 213
- updateLS, 249
- DataBase_, 29
- DataBase_
 - id, 29
 - linkDstVec, 30
 - linkSrcVec, 30
 - linkTab, 30
 - lspVec, 30
 - nbLinks, 30
 - nbNodes, 30
 - nodeVec, 30
- database_api.h, 250
 - DBaddLink, 251
 - DBaddLSP, 253
 - DBaddNode, 258
 - DBdestroy, 258
 - DBevalLSOnRemove, 259
 - DBevalLSOnSetup, 259
 - DBgetID, 259
 - DBgetLinkDst, 260
 - DBgetLinkID, 260
 - DBgetLinkLSPs, 261
 - DBgetLinkSrc, 261
 - DBgetLinkState, 261
 - DBgetLSP, 262
 - DBgetMaxNodeID, 262
 - DBgetNbLinks, 263
 - DBgetNbNodes, 263
 - DBgetNodeInNeighb, 263
 - DBgetNodeOutNeighb, 264
 - DBlinkStateCopy, 264
 - DBlinkStateDestroy, 265
 - DBlinkStateEnd, 265
 - DBlinkStateInit, 266
 - DBlinkStateNew, 267
 - DBlspCompare, 269
 - DBlspCopy, 269
 - DBlspDestroy, 270
 - DBlspEnd, 270
 - DBlspInit, 271
 - DBlspListDestroy, 272
 - DBlspListEnd, 272
 - DBlspListInit, 272
 - DBlspListInsert, 273
 - DBlspListNew, 274
 - DBlspListRemove, 275
 - DBlspNew, 276
 - DBnew, 277
 - DBprintDB, 278
 - DBremoveLink, 279
 - DBremoveLSP, 279
 - DBremoveNode, 281
 - DBsetLinkState, 282
 - DBupdateLSOnRemove, 283
 - DBupdateLSOnSetup, 283
- database_st.h
 - GLOBAL_BACK, 285
 - LOCAL_BACK, 285
 - PRIM, 285
- database_st.h, 284
 - DataBase, 285
 - DBLabelSwitchedPath, 285
 - DBLinkState, 285
 - DBLSPList, 285
 - DBLSPType, 285
- database_util.c, 287
 - DBlinkDestroy, 288
 - DBlinkEnd, 289
 - DBlinkInit, 289
 - DBlinkNew, 289
 - DBlinkTabDestroy, 290
 - DBlinkTabEnd, 291
 - DBlinkTabInit, 291
 - DBlinkTabNew, 292
 - DBlinkTabRemove, 293
 - DBlinkTabResize, 293
 - DBlinkTabSet, 295
 - DBlspVecDestroy, 295
 - DBlspVecEnd, 296
 - DBlspVecInit, 296
 - DBlspVecNew, 297
 - DBlspVecRemove, 297
 - DBlspVecResize, 298
 - DBlspVecSet, 298
 - DBnodeDestroy, 299
 - DBnodeEnd, 299
 - DBnodeInit, 300
 - DBnodeNew, 300
 - DBnodeVecDestroy, 301
 - DBnodeVecEnd, 302
 - DBnodeVecInit, 302
 - DBnodeVecNew, 303
 - DBnodeVecRemove, 303
 - DBnodeVecResize, 304
 - DBnodeVecSet, 305
 - DBprintLink, 305
 - DBprintNode, 306
- database_util.h, 308
 - DBLink, 310
 - DBlinkDestroy, 310

- DBlinkEnd, [311](#)
- DBlinkInit, [311](#)
- DBlinkNew, [311](#)
- DBLinkTab, [310](#)
- DBlinkTabDestroy, [312](#)
- DBlinkTabEnd, [312](#)
- DBlinkTabGet, [310](#)
- DBlinkTabInit, [313](#)
- DBlinkTabNew, [314](#)
- DBlinkTabRemove, [315](#)
- DBlinkTabResize, [315](#)
- DBlinkTabSet, [316](#)
- DBLSPVec, [310](#)
- DBlspVecDestroy, [317](#)
- DBlspVecEnd, [317](#)
- DBlspVecGet, [310](#)
- DBlspVecInit, [318](#)
- DBlspVecNew, [319](#)
- DBlspVecRemove, [319](#)
- DBlspVecResize, [319](#)
- DBlspVecSet, [320](#)
- DBNode, [310](#)
- DBnodeDestroy, [321](#)
- DBnodeEnd, [321](#)
- DBnodeInit, [321](#)
- DBnodeNew, [322](#)
- DBNodeVec, [310](#)
- DBnodeVecDestroy, [323](#)
- DBnodeVecEnd, [323](#)
- DBnodeVecGet, [310](#)
- DBnodeVecInit, [323](#)
- DBnodeVecNew, [324](#)
- DBnodeVecRemove, [325](#)
- DBnodeVecResize, [325](#)
- DBnodeVecSet, [326](#)
- DBprintLink, [327](#)
- DBprintNode, [328](#)
- DBaddLink
 - [database-oli.c, 175](#)
 - [database.c, 216](#)
 - [database_api.h, 251](#)
- DBaddLSP
 - [database-oli.c, 176](#)
 - [database.c, 217](#)
 - [database_api.h, 253](#)
- DBaddNode
 - [database-oli.c, 181](#)
 - [database.c, 222](#)
 - [database_api.h, 258](#)
- DBdestroy
 - [database-oli.c, 182](#)
 - [database.c, 223](#)
 - [database_api.h, 258](#)
- DBevallLSONRemove
 - [database-oli.c, 182](#)
 - [database.c, 224](#)
 - [database_api.h, 259](#)
- DBevallLSONSetup
 - [database-oli.c, 182](#)
 - [database.c, 224](#)
 - [database_api.h, 259](#)
- DBgetID
 - [database-oli.c, 182](#)
 - [database.c, 224](#)
 - [database_api.h, 259](#)
- DBgetLinkDst
 - [database-oli.c, 183](#)
 - [database.c, 224](#)
 - [database_api.h, 260](#)
- DBgetLinkID
 - [database-oli.c, 183](#)
 - [database.c, 225](#)
 - [database_api.h, 260](#)
- DBgetLinkLSPs
 - [database-oli.c, 183](#)
 - [database.c, 225](#)
 - [database_api.h, 261](#)
- DBgetLinkSrc
 - [database-oli.c, 184](#)
 - [database.c, 226](#)
 - [database_api.h, 261](#)
- DBgetLinkState
 - [database-oli.c, 184](#)
 - [database.c, 226](#)
 - [database_api.h, 261](#)
- DBgetLSP
 - [database-oli.c, 185](#)
 - [database.c, 227](#)
 - [database_api.h, 262](#)
- DBgetMaxNodeID
 - [database-oli.c, 185](#)
 - [database.c, 227](#)
 - [database_api.h, 262](#)
- DBgetNbLinks
 - [database-oli.c, 185](#)
 - [database.c, 227](#)
 - [database_api.h, 263](#)
- DBgetNbNodes
 - [database-oli.c, 185](#)
 - [database.c, 228](#)
 - [database_api.h, 263](#)
- DBgetNodeInNeighb
 - [database-oli.c, 186](#)
 - [database.c, 228](#)
 - [database_api.h, 263](#)
- DBgetNodeOutNeighb
 - [database-oli.c, 186](#)
 - [database.c, 228](#)

- database_api.h, 264
- DBLabelSwitchedPath
 - database_st.h, 285
- DBLabelSwitchedPath_, 32
- DBLabelSwitchedPath_
 - backLSPIDs, 33
 - bw, 33
 - forbidLinks, 33
 - id, 33
 - noContentionId, 33
 - path, 33
 - precedence, 34
 - primID, 34
 - primPath, 34
 - type, 34
- DBLink
 - database_util.h, 310
- DBLink_, 35
 - id, 35
 - lspList, 35
 - state, 35
- DBlinkDestroy
 - database_util.c, 288
 - database_util.h, 310
- DBlinkEnd
 - database_util.c, 289
 - database_util.h, 311
- DBlinkInit
 - database_util.c, 289
 - database_util.h, 311
- DBlinkNew
 - database_util.c, 289
 - database_util.h, 311
- DBLinkState
 - database_st.h, 285
- DBLinkState_, 37
- DBLinkState_
 - bbw, 38
 - cap, 38
 - color, 38
 - fbw, 38
 - pbw, 38
 - rbw, 38
 - remoteBbw, 39
 - remoteFbw, 39
- DBlinkStateCopy
 - database-oli.c, 187
 - database.c, 229
 - database_api.h, 264
- DBlinkStateDestroy
 - database-oli.c, 187
 - database.c, 229
 - database_api.h, 265
- DBlinkStateEnd
 - database-oli.c, 188
 - database.c, 230
 - database_api.h, 265
- DBlinkStateInit
 - database-oli.c, 188
 - database.c, 230
 - database_api.h, 266
- DBlinkStateNew
 - database-oli.c, 189
 - database.c, 232
 - database_api.h, 267
- DBLinkTab
 - database_util.h, 310
- DBLinkTab_, 40
- DBLinkTab_
 - cont, 40
 - size, 40
- DBlinkTabDestroy
 - database_util.c, 290
 - database_util.h, 312
- DBlinkTabEnd
 - database_util.c, 291
 - database_util.h, 312
- DBlinkTabGet
 - database_util.h, 310
- DBlinkTabInit
 - database_util.c, 291
 - database_util.h, 313
- DBlinkTabNew
 - database_util.c, 292
 - database_util.h, 314
- DBlinkTabRemove
 - database_util.c, 293
 - database_util.h, 315
- DBlinkTabResize
 - database_util.c, 293
 - database_util.h, 315
- DBlinkTabSet
 - database_util.c, 295
 - database_util.h, 316
- DBlspCompare
 - database-oli.c, 191
 - database.c, 233
 - database_api.h, 269
- DBlspCopy
 - database-oli.c, 191
 - database.c, 234
 - database_api.h, 269
- DBlspDestroy
 - database-oli.c, 192
 - database.c, 234
 - database_api.h, 270
- DBlspEnd
 - database-oli.c, 192

- database.c, 235
- database_api.h, 270
- DBlspInit
 - database-oli.c, 192
 - database.c, 235
 - database_api.h, 271
- DBLSPList
 - database_st.h, 285
- DBLSPList_, 41
 - cont, 41
 - size, 41
 - top, 41
- DBlspListDestroy
 - database-oli.c, 193
 - database.c, 236
 - database_api.h, 272
- DBlspListEnd
 - database-oli.c, 194
 - database.c, 237
 - database_api.h, 272
- DBlspListInit
 - database-oli.c, 194
 - database.c, 237
 - database_api.h, 272
- DBlspListInsert
 - database-oli.c, 194
 - database.c, 237
 - database_api.h, 273
- DBlspListNew
 - database-oli.c, 196
 - database.c, 239
 - database_api.h, 274
- DBlspListRemove
 - database-oli.c, 196
 - database.c, 239
 - database_api.h, 275
- DBlspNew
 - database-oli.c, 197
 - database.c, 240
 - database_api.h, 276
- DBLSPType
 - database_st.h, 285
- DBLSPVec
 - database_util.h, 310
- DBLSPVec_, 42
 - cont, 42
 - size, 42
- DBlspVecDestroy
 - database_util.c, 295
 - database_util.h, 317
- DBlspVecEnd
 - database_util.c, 296
 - database_util.h, 317
- DBlspVecGet
 - database_util.h, 310
- DBlspVecInit
 - database_util.c, 296
 - database_util.h, 318
- DBlspVecNew
 - database_util.c, 297
 - database_util.h, 319
- DBlspVecRemove
 - database_util.c, 297
 - database_util.h, 319
- DBlspVecResize
 - database_util.c, 298
 - database_util.h, 319
- DBlspVecSet
 - database_util.c, 298
 - database_util.h, 320
- DBLVEC_INITSIZE
 - setup.h, 430
- dblVecCopy
 - common.c, 120
 - common.h, 140
- dblVecDestroy
 - common.c, 120
 - common.h, 140
- dblVecEnd
 - common.c, 121
 - common.h, 141
- dblVecGet
 - common.c, 121
 - common.h, 141
- dblVecInit
 - common.c, 121
 - common.h, 141
- dblVecNew
 - common.c, 122
 - common.h, 142
- dblVecPopBack
 - common.c, 123
 - common.h, 143
- dblVecPushBack
 - common.c, 123
 - common.h, 143
- dblVecResize
 - common.c, 124
 - common.h, 144
- dblVecSet
 - common.c, 124
 - common.h, 144
- DBnew
 - database-oli.c, 198
 - database.c, 241
 - database_api.h, 277
- DBNode
 - database_util.h, 310

- DBNode_, 43
 - id, 43
 - inNeighb, 43
 - outNeighb, 43
- DBnodeDestroy
 - database_util.c, 299
 - database_util.h, 321
- DBnodeEnd
 - database_util.c, 299
 - database_util.h, 321
- DBnodeInit
 - database_util.c, 300
 - database_util.h, 321
- DBnodeNew
 - database_util.c, 300
 - database_util.h, 322
- DBNodeVec
 - database_util.h, 310
- DBNodeVec_, 44
- DBNodeVec_
 - cont, 44
 - size, 44
 - top, 44
- DBnodeVecDestroy
 - database_util.c, 301
 - database_util.h, 323
- DBnodeVecEnd
 - database_util.c, 302
 - database_util.h, 323
- DBnodeVecGet
 - database_util.h, 310
- DBnodeVecInit
 - database_util.c, 302
 - database_util.h, 323
- DBnodeVecNew
 - database_util.c, 303
 - database_util.h, 324
- DBnodeVecRemove
 - database_util.c, 303
 - database_util.h, 325
- DBnodeVecResize
 - database_util.c, 304
 - database_util.h, 325
- DBnodeVecSet
 - database_util.c, 305
 - database_util.h, 326
- DBprintDB
 - database-oli.c, 199
 - database.c, 242
 - database_api.h, 278
- DBprintLink
 - database_util.c, 305
 - database_util.h, 327
- DBprintNode
 - database_util.c, 306
 - database_util.h, 328
- DBremoveLink
 - database-oli.c, 200
 - database.c, 243
 - database_api.h, 279
- DBremoveLSP
 - database-oli.c, 201
 - database.c, 244
 - database_api.h, 279
- DBremoveNode
 - database-oli.c, 202
 - database.c, 246
 - database_api.h, 281
- DBsetLinkState
 - database-oli.c, 203
 - database.c, 247
 - database_api.h, 282
- DBupdateLSPOnRemove
 - database-oli.c, 204
 - database.c, 247
 - database_api.h, 283
- DBupdateLSPOnSetup
 - database-oli.c, 204
 - database.c, 247
 - database_api.h, 283
- delay
 - PrimaryComputationConfig_, 57
- DIGIT_PRECISION
 - setup.h, 430
- dijkstra.c, 329
 - CPdestroyPQ, 329
 - CPdestroyTN, 330
 - CPendPQ, 330
 - CPinitPQ, 330
 - CPinsertPQ, 331
 - CPnewPQ, 332
 - CPnewTN, 332
 - CPpopTop, 333
- dijkstra.h, 335
 - CPdestroyPQ, 336
 - CPdestroyTN, 336
 - CPDijkNode, 336
 - CPendPQ, 337
 - CPinitPQ, 337
 - CPinsertPQ, 337
 - CPnewPQ, 339
 - CPpopTop, 339
 - CPPrioQueue, 336
 - CPTreeNode, 336
- DoubleVec
 - common.h, 139
- DoubleVec_, 45
- DoubleVec_

- cont, [45](#)
 - size, [45](#)
 - top, [45](#)
- dst
 - LSPrerouteInfo_, [54](#)
- endTopo
 - primaryPath.c, [377](#)
 - primaryPath_util.h, [410](#)
- error.c, [341](#)
 - addError, [341](#)
 - errorDestroy, [343](#)
 - ErrorElem, [341](#)
 - errorInit, [343](#)
 - errorManager, [344](#)
 - printErrorStack, [343](#)
- error.h, [345](#)
 - addError, [346](#)
 - CRITICAL, [346](#)
 - errorDestroy, [348](#)
 - errorInit, [348](#)
 - ErrorList, [346](#)
 - errorManager, [349](#)
 - GravityLevel, [346](#)
 - INFO, [346](#)
 - PANIC, [346](#)
 - printErrorStack, [348](#)
 - WARNING, [346](#)
- ERROR_PROVISION
 - setup.h, [430](#)
- errorDestroy
 - error.c, [343](#)
 - error.h, [348](#)
- ErrorElem
 - error.c, [341](#)
- ErrorElem_, [46](#)
- ErrorElem_
 - gravity, [46](#)
 - message, [46](#)
- errorInit
 - error.c, [343](#)
 - error.h, [348](#)
- ErrorList
 - error.h, [346](#)
- ErrorList_, [47](#)
- ErrorList_
 - list, [47](#)
 - size, [47](#)
 - top, [47](#)
- ERRORLIST_INITSIZE
 - setup.h, [431](#)
- errorManager
 - error.c, [344](#)
 - error.h, [349](#)
- ERRORMSG_SIZE
 - setup.h, [431](#)
- evalLS
 - database-oli.c, [204](#)
 - database.c, [248](#)
- FALSE
 - common.h, [139](#)
- father
 - CPTreeNode_, [26](#)
- fbw
 - DBLinkState_, [38](#)
- fillTopo
 - primaryPath.c, [378](#)
 - primaryPath_util.h, [410](#)
- forbidLinks
 - DBLabelSwitchedPath_, [33](#)
 - LSPRequest_, [51](#)
- free
 - common.h, [137](#)
- from
 - CPDijkNode_, [22](#)
- gain
 - BKConnectInfo_, [12](#)
- getRequestDst
 - primaryPath.c, [382](#)
 - primaryPath_util.h, [414](#)
- getRequestSrc
 - primaryPath.c, [382](#)
 - primaryPath_util.h, [415](#)
- GLOBAL
 - backup_st.h, [118](#)
- GLOBAL_BACK
 - database_st.h, [285](#)
- gravity
 - ErrorElem_, [46](#)
- GravityLevel
 - error.h, [346](#)
- gt
 - CPTreeNode_, [26](#)
- id
 - DataBase_, [29](#)
 - DBLabelSwitchedPath_, [33](#)
 - DBLink_, [35](#)
 - DBNode_, [43](#)
 - LSPRequest_, [51](#)
 - LSPrerouteInfo_, [54](#)
- INFO
 - error.h, [346](#)
- info
 - BKConnect_, [11](#)
 - BKNode_, [15](#)

- initScore
 - primaryPath.c, 383
 - primaryPath_util.h, 415
- initTopo
 - primaryPath.c, 384
 - primaryPath_util.h, 416
- inNeighb
 - BKNode_, 15
 - DBNode_, 43
- isValidLSPLink
 - predicate.c, 351
 - predicate.h, 355
- isValidRequestLink
 - predicate.c, 352
 - predicate.h, 356
- key
 - CPTreeNode_, 26
- leq
 - CPTreeNode_, 26
- libavl_allocator, 48
 - libavl_free, 48
 - libavl_malloc, 48
- libavl_free
 - libavl_allocator, 48
- libavl_malloc
 - libavl_allocator, 48
- linkDstVec
 - DataBase_, 30
- linkSrcVec
 - DataBase_, 30
- linkState
 - BKConnect_, 11
- linkTab
 - DataBase_, 30
- LINKTAB_INITSIZE
 - setup.h, 431
- LINUX
 - setup.h, 431
- list
 - ErrorList_, 47
- load
 - PrimaryComputationConfig_, 57
- loadBal
 - PrimaryComputationConfig_, 57
- LOCAL
 - backup_st.h, 118
- LOCAL.BACK
 - database_st.h, 285
- longCompare
 - common.c, 125
- LongList
 - common.h, 139
- longListCopy
 - common.h, 137
- longListDestroy
 - common.h, 138
- longListEnd
 - common.h, 138
- longListInit
 - common.h, 138
- longListInsert
 - common.c, 125
 - common.h, 145
- longListMerge
 - common.c, 126
 - common.h, 145
- longListNew
 - common.h, 138
- longListPopBack
 - common.h, 138
- longListPushBack
 - common.h, 138
- longListRemove
 - common.c, 127
 - common.h, 147
- longListResize
 - common.h, 138
- longListSort
 - common.c, 127
 - common.h, 147
- LongVec
 - common.h, 139
- LongVec_, 49
- LongVec_
 - cont, 49
 - size, 49
 - top, 49
- LONGVEC_INITSIZE
 - setup.h, 431
- longVecCopy
 - common.c, 128
 - common.h, 148
- longVecDestroy
 - common.c, 128
 - common.h, 148
- longVecEnd
 - common.c, 129
 - common.h, 148
- longVecGet
 - common.c, 129
 - common.h, 149
- longVecInit
 - common.c, 130
 - common.h, 149
- longVecNew
 - common.c, 130

- common.h, 150
- longVecPopBack
 - common.c, 131
 - common.h, 151
- longVecPushBack
 - common.c, 131
 - common.h, 151
- longVecResize
 - common.c, 132
 - common.h, 152
- longVecSet
 - common.c, 132
 - common.h, 152
- lspList
 - DBLink_, 35
- LSPLIST_INITSIZE
 - setup.h, 431
- LSPREQLIST_INITSIZE
 - setup.h, 431
- LSPRequest
 - computation.st.h, 170
- LSPRequest_, 50
 - bw, 51
 - forbidLinks, 51
 - id, 51
 - path, 51
 - precedence, 51
 - primID, 51
 - rerouteInfo, 52
 - type, 52
- lspRequestCopy
 - computation.c, 155
 - computation.api.h, 163
- lspRequestDestroy
 - computation.c, 156
 - computation.api.h, 163
- lspRequestEnd
 - computation.c, 156
 - computation.api.h, 164
- lspRequestInit
 - computation.c, 157
 - computation.api.h, 164
- LSPRequestList
 - computation.st.h, 170
- LSPRequestList_, 53
- LSPRequestList_
 - cont, 53
 - size, 53
- lspRequestListEnd
 - computation.c, 157
 - computation.api.h, 165
- lspRequestListGet
 - computation.c, 158
 - computation.api.h, 165
- lspRequestListInit
 - computation.c, 158
 - computation.api.h, 165
- lspRequestListResize
 - computation.c, 159
 - computation.api.h, 166
- lspRequestListSet
 - computation.c, 160
 - computation.api.h, 167
- lspRequestListSize
 - computation.c, 160
- lspRequestListsize
 - computation.api.h, 167
- lspRequestNew
 - computation.c, 160
 - computation.api.h, 167
- LSPrerouteInfo
 - computation.st.h, 170
- LSPrerouteInfo_, 54
- LSPrerouteInfo_
 - dst, 54
 - id, 54
 - src, 54
- lspVec
 - DataBase_, 30
- LSPVEC_INITSIZE
 - setup.h, 431
- makeRerouteScore
 - rerouting.c, 424
 - rerouting.h, 427
- makeScore
 - primaryPath.c, 384
 - primaryPath_util.h, 417
- malloc
 - common.h, 138
- marked
 - CPDijkNode_, 22
- max
 - common.h, 139
- message
 - ErrorElem_, 46
- min
 - common.h, 139
- mycalloc
 - common.c, 133
 - common.h, 153
- myfree
 - common.c, 133
 - common.h, 153
- mymalloc
 - common.c, 134
 - common.h, 153
- myrealloc

- common.c, 134
- common.h, 154
- NB_OA
 - setup.h, 431
- NB_PREEMPTION
 - setup.h, 432
- nbLinks
 - BKTopology_, 20
 - DataBase_, 30
- nbNodes
 - BKTopology_, 20
 - DataBase_, 30
- neighbId
 - BKConnect_, 11
- neighbInd
 - BKNode_, 15
- newCost
 - BKNodeInfo_, 17
- newNeighbInd
 - BKNodeInfo_, 17
- newSum
 - BKNodeInfo_, 17
- noContentionId
 - DBLabelSwitchedPath_, 33
- node
 - CPDijkNode_, 22
 - CPTreeNode_, 27
- nodeId
 - BKNode_, 16
- nodeInd
 - BKTopology_, 20
- nodeVec
 - BKTopology_, 21
 - DataBase_, 30
- NODEVEC_INITSIZE
 - setup.h, 432
- noLoop
 - primaryPath.c, 386
 - primaryPath_util.h, 418
- NONE
 - backup_st.h, 118
- operation
 - database-oli.c, 172
 - database.c, 213
- operation_
 - database-oli.c, 172
 - database.c, 213
- outNeighb
 - BKNode_, 16
 - DBNode_, 43
- PANIC
 - error.h, 346
- path
 - DBLabelSwitchedPath_, 33
 - LSPRequest_, 51
- pbw
 - DBLinkState_, 38
- precedence
 - DBLabelSwitchedPath_, 34
 - LSPRequest_, 51
- predicate.c, 350
 - capacityClause, 350
 - colorClause, 351
 - isValidLSPLink, 351
 - isValidRequestLink, 352
- predicate.h, 355
 - isValidLSPLink, 355
 - isValidRequestLink, 356
- PredicateConfig
 - setup.h, 432
- predicateConfig
 - DAMOTEConfig_, 28
- PredicateConfig_, 56
- PredicateConfig_
 - allowReroute, 56
 - capacityClause, 56
 - colorClause, 56
- PRIM
 - database_st.h, 285
- PrimaryComputationConfig
 - setup.h, 432
- primaryComputationConfig
 - DAMOTEConfig_, 28
- PrimaryComputationConfig_, 57
- PrimaryComputationConfig_
 - delay, 57
 - load, 57
 - loadBal, 57
 - relLoad, 57
 - rerouting, 57
 - sqLoad, 57
 - sqRelLoad, 57
- primaryPath.c, 359
- primaryPath.c
 - activateNodeInfo, 360
 - bellmanKalaba, 360
 - bkConnectVecCopy, 364
 - bkConnectVecDestroy, 365
 - bkConnectVecEnd, 365
 - bkConnectVecGet, 366
 - bkConnectVecInit, 366
 - bkConnectVecNew, 367
 - bkConnectVecPopBack, 367
 - bkConnectVecPushBack, 368
 - bkConnectVecResize, 368

- bkConnectVecSet, 369
- bkNodeVecDestroy, 369
- bkNodeVecEnd, 370
- bkNodeVecGet, 370
- bkNodeVecInit, 371
- bkNodeVecNew, 372
- bkNodeVecPopBack, 373
- bkNodeVecPushBack, 374
- bkNodeVecResize, 374
- bkNodeVecSet, 375
- computePrimaryPath, 376
- endTopo, 377
- fillTopo, 378
- getRequestDst, 382
- getRequestSrc, 382
- initScore, 383
- initTopo, 384
- makeScore, 384
- noLoop, 386
- printTopo, 386
- updateNodeInfoOnElect, 387
- updateRequest, 387
- primaryPath_api.h, 390
- primaryPath_api.h
 - computePrimaryPath, 390
- primaryPath_util.h, 392
- primaryPath_util.h
 - activateNodeInfo, 394
 - bellmanKalaba, 394
 - BKConnect, 393
 - BKConnectInfo, 393
 - BKConnectVec, 394
 - bkConnectVecCopy, 398
 - bkConnectVecDestroy, 399
 - bkConnectVecEnd, 399
 - bkConnectVecGet, 399
 - bkConnectVecInit, 400
 - bkConnectVecNew, 401
 - bkConnectVecPopBack, 401
 - bkConnectVecPushBack, 402
 - bkConnectVecResize, 402
 - bkConnectVecSet, 403
 - BKNode, 394
 - BKNodeInfo, 394
 - BKNodeVec, 394
 - bkNodeVecDestroy, 403
 - bkNodeVecEnd, 404
 - bkNodeVecGet, 404
 - bkNodeVecInit, 405
 - bkNodeVecNew, 406
 - bkNodeVecPopBack, 407
 - bkNodeVecPushBack, 408
 - bkNodeVecResize, 408
 - bkNodeVecSet, 409
 - BKTopology, 394
 - endTopo, 410
 - fillTopo, 410
 - getRequestDst, 414
 - getRequestSrc, 415
 - initScore, 415
 - initTopo, 416
 - makeScore, 417
 - noLoop, 418
 - printTopo, 419
 - updateNodeInfoOnElect, 419
 - updateRequest, 420
 - primID
 - DBLabelSwitchedPath_, 34
 - LSPRequest_, 51
 - primPath
 - DBLabelSwitchedPath_, 34
 - printErrorStack
 - error.c, 343
 - error.h, 348
 - printTopo
 - primaryPath.c, 386
 - primaryPath_util.h, 419
 - rbw
 - DBLinkState_, 38
 - realloc
 - common.h, 139
 - relLoad
 - PrimaryComputationConfig_, 57
 - remoteBbw
 - DBLinkState_, 39
 - remoteFbw
 - DBLinkState_, 39
 - REMOVE
 - database-oli.c, 172
 - database.c, 213
 - rerouteInfo
 - LSPRequest_, 52
 - rerouting
 - PrimaryComputationConfig_, 57
 - rerouting.c, 422
 - chooseReroutedLSPs, 422
 - makeRerouteScore, 424
 - rerouting.h, 425
 - chooseReroutedLSPs, 425
 - makeRerouteScore, 427
 - ReroutingConfig
 - setup.h, 432
 - reroutingConfig
 - DAMOTEConfig_, 28
 - ReroutingConfig_, 59
 - ReroutingConfig_
 - scoreCoef, 59

- root
 - CPPrioQueue_, 24
- scoreCoef
 - ReroutingConfig_, 59
- SETUP
 - database-oli.c, 172
 - database.c, 213
- setup.c, 428
 - damoteConfig, 428
- setup.h, 429
 - BKCONNECTVEC.INITSIZE, 430
 - BKNODEVEC.INITSIZE, 430
 - CONTAINER_TEST, 430
 - DAMOTEConfig, 432
 - damoteConfig, 432
 - DBLVEC.INITSIZE, 430
 - DIGIT_PRECISION, 430
 - ERROR_PROVISION, 430
 - ERRORLIST.INITSIZE, 431
 - ERRORMSG_SIZE, 431
 - LINKTAB.INITSIZE, 431
 - LINUX, 431
 - LONGVEC.INITSIZE, 431
 - LSPLIST.INITSIZE, 431
 - LSPREQLIST.INITSIZE, 431
 - LSPVEC.INITSIZE, 431
 - NB_OA, 431
 - NB_PREEMPTION, 432
 - NODEVEC.INITSIZE, 432
 - PredicateConfig, 432
 - PrimaryComputationConfig, 432
 - ReroutingConfig, 432
 - SIMULATOR, 432
- SIMULATOR
 - setup.h, 432
- size
 - BKConnectVec_, 13
 - BKNodeVec_, 18
 - CPPrioQueue_, 24
 - DBLinkTab_, 40
 - DBLSPList_, 41
 - DBLSPVec_, 42
 - DBNodeVec_, 44
 - DoubleVec_, 45
 - ErrorList_, 47
 - LongVec_, 49
 - LSPRequestList_, 53
- sqLoad
 - PrimaryComputationConfig_, 57
- sqRelLoad
 - PrimaryComputationConfig_, 57
- src
 - LSPrerouteInfo_, 54
- state
 - DBLink_, 35
- sum
 - BKNodeInfo_, 17
- top
 - BKConnectVec_, 13
 - BKNodeVec_, 18
 - CPPrioQueue_, 24
 - DBLSPList_, 41
 - DBNodeVec_, 44
 - DoubleVec_, 45
 - ErrorList_, 47
 - LongVec_, 49
- TRUE
 - common.h, 139
- type
 - DBLabelSwitchedPath_, 34
 - LSPRequest_, 52
- updateLS
 - database-oli.c, 206
 - database.c, 249
- updateNodeInfoOnElect
 - primaryPath.c, 387
 - primaryPath_util.h, 419
- updateRequest
 - primaryPath.c, 387
 - primaryPath_util.h, 420
- val
 - CPDijkNode_, 22
- WARNING
 - error.h, 346