# SPACE EXTRACTION APPLICATION NOTE ABOUT NODES

*S. de Graaf*

Circuits and Systems Group
Faculty of Electrical Engineering,
Mathematics and Computer Science
Delft University of Technology
The Netherlands

Report EWI-ENS 03-05
October 10, 2003



Last revision: November 4, 2003.

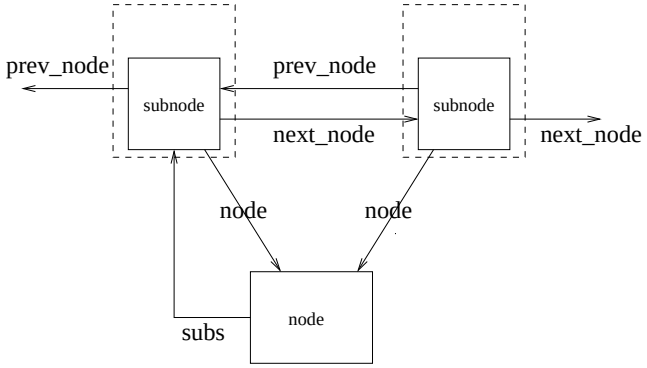
## 1. INTRODUCTION

This application note describes everything what space programmers must know about the nodes in space. Nodes are conducting layout area's and do not have resistance. Only conducting layers (layout masks) can be nodes. The via (contact) masks do not need to be defined as conductors. The via masks are help masks, thus the contact between conductors can be found. The contact itself can have a resistance value. The conductors have also a sheet resistance value (>= 0 ohm). The nodes are translated to nets in the circuit view.

There is one special mask (@sub), that is predefined as conductor mask. This mask has also a conductor number (nrOfConductors - 1) and can have a color. If the mask is specified in the technology file, the mask index must be (nrOfMasks - 1). The mask index -1 is used in the space program, to make distinction between this conductor and interconnect masks conductors.

## 2. SUBNODES

The same node can be shared by a number of subnodes.
The following figure explains the connections between the node/subnode data structures.




The subnodes are located in the "tile_t" or "nodePoint_t" data struct. The "nodePoint_t" is used in the extract pass by interconnect/substrate res extraction. In that case, the "tile_t" data is not used (but exists). Both contain an array of subnode pointers as well (member "cons" with size nrOfConductors), one for each possible conductor. The pointers are initialized to NULL (see createTile and createNodePoint). Note that createNodePoint sets the pointers to the subnode data itself, because there is given an array of existing conductors as argument. For "tile_t", this is done with macro CONS for each found conductor (see enumPair). It is important to note, that after createNodePoint the subnode data still needs to be initialized (subnodeNew or subnodeCopy must be done).

After the pointer is set, the "subnode_t" data needs to be initialized. Note that a subnode must always point to a node. Function subnodeNew or subnodeCopy (see "lump/lump.c") must initialize the subnode. The subnode data in the tiles and nodePoints is never disposed. Note, when tiles or nodePoints are disposed, they are put in a free list.

However, there are also substrate subnodes. Each tile that has a substrate terminal reference, has a pointer "subcont", that points to "subcontRef_t" data. This "subcontRef_t" contains a pointer "subn" to an allocated subnode. Note that the subcontRef's (one or more) also point to "subcont_t" data, which also contains pointers to allocated subnodes. These subnodes must separately be disposed (see function subContDel).

Note that spiders also points to subnodes, when a mesh is generated (options **-B** and/or **-3C**). These subnode pointers are only used to put the calculated capacitance or resistance values between the nodes. Function spiderNew assigns tile or nodePoint or substrate subnodes to the spiders.

At last, there are some other subnodes. Transistors have d/s boundaries, each boundary has a "dsCond" pointer to an allocated subnode. There are also temporary subnodes used

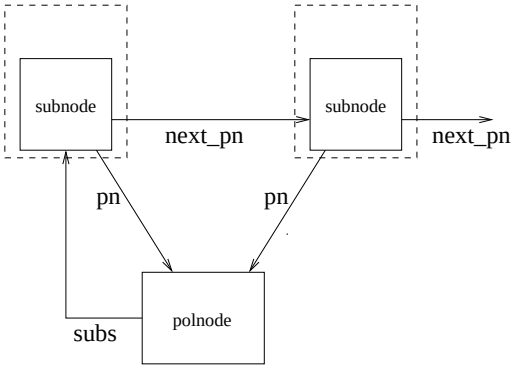
to add conductors with a new node in a group. There is also a subnode array (subNODES) used to set the initial subnodes by createNodePoint.

### 2.1 Subnode Data Struct

The "subnode_t" data struct contains the following members (see "include/subnode.h"):

```
typedef struct subnode {
    struct Node    * node;
    struct polnode * pn;        // BIPOLAR
    struct subnode * next_pn;   // BIPOLAR
    struct subnode * next_node;   /* next for sub node */
    struct subnode * prev_node;   /* next for sub node */
} subnode_t;
```

As shown above, the subnode can point to a node (member "node"). The node itself points back (with member "subs") to a double linked list of subnodes. The subnode members "prev_node" and "next_node" are used for this double linked list. The subnode can also point to a polnode with "pn". The polnode itself points back (with member "subs") to a linked list of subnodes. The subnode member "next_pn" is used for this linked list. Note that polnodes only are used, when there are bipolar elements in the technology file (for more details see the polnode section).




### 2.2 Function: subnodeNew

This function has a subnode as argument and initializes this subnode. Note that each subnode needs to point to a node. Only this function does a call to function createNode. Function subnodeNew initializes also the group part of the node, it calls NEW to create a group_t data struct. Group data structs are only created by subnodeNew. Some node members are outside subnodeNew initialized.

### 2.3 Function: subnodeCopy

This function has two subnode arguments (subnA and subnB). The function does not create a new node for subnode subnB, but lets subnB points to the node of subnode subnA. Note that subnB may not have a node, because in that case subnodeJoin needs to be used. Subnode subnB becomes the head of the subnode linked list of the node.

Subnode subnA is also somewhere in this list. The subnodeCopy function is done for two cases, (1) for new tile/nodepoint subnodes (in place of subnodeNew/subnodeJoin), and (2) for duplicate subnodes (for example the dsCond of a transistor), which must point to the same node/polnode.

### 2.4 Function: subnodeJoin

This function has two subnode arguments (subnA and subnB). When both subnodes have different nodes, these nodes are joined by nodeJoin. When both subnodes have different polnodes, these polnodes are joined by polnodeJoin.

### 2.5 Function: subnodeDel

This function has a subnode as argument and deletes this subnode. When the subnode has a polnode, the subnode is removed from the subnode list of the polnode. When it was the last subnode of the list, the polnode is deleted by polnodeDel. The subnode is also removed from the subnode list of the node. When it was the last subnode of the list, the node is ready and readyNode is called. Note that the node must have a subnode reference (node->subs).

### 2.6 Variable: noSubnodeJoin

This variable is default always false, but can get true, when both the special options **-K** (optNoCore) and **-M** (optReadNetTerm) are used. See "lump/lump.h", macro EnableNoSubnodeJoin sets the variable to true. The macro CheckNoSubnodeJoin can also set the variable, when the tiles have unequal "instNr". The variable noSubnodeJoin is used in functions subnodeCopy and subnodeJoin (to forbit the copy and join) and is there reset. After setting noSubnodeJoin one of these functions must be called.

### 2.7 Variable: inSubnod

Integer count, initialized to zero before each pass by initLump. Used for extraction statistics (printed with option **-i**). Tells, howmany subnodeNew and subnodeCopy functions are done. Note that the count for subnodeNew is incremented by createNode.

### 2.8 Variable: currIntSubnod

Integer count, initialized to zero before each pass by initLump. Used for extraction statistics (printed with option **-i**). Tells, howmany subnodeNew and subnodeCopy functions are done (see inSubnod) and are remaining after subnodeDel. Function subnodeDel decrements currIntSubnod. The result should be zero.

### 2.9 Variable: maxIntSubnod

Integer count, initialized to zero before each pass by initLump. Used for extraction statistics (printed with option **-i**). The maximum internal value of currIntSubnod (see above) in the pass.

## 3. NODE FUNCTIONS

### 3.1 Function: createNode

The createNode function shall first try to reuse disposed nodes, which are in the "free_list". This function is only called by function subnodeNew. Function subnodeNew initialize some members of the node (the node group part).

### 3.2 Function: disposeNode

This function is only called by findRoot. It disposes the condemned node. It disposes also existing node moments with function disposeMoments. This moments pointer needs to be initialized to NULL in createNode. The node is put into the node "free_list". The node "next" pointer is used for this free linked list.

### 3.3 Function: nodeJoin

The function has two arguments, two node pointers (nA and nB). First it decides, out of efficiency reasons, which node must be nA and which nB. It joins nB with nA (nB != nA), and node nB is condemned (by nodeDel). The not condemned node nA is returned. All parts of nB are moved to nA (flag settings, conductors, capacitors, moments, subnode refs, bjt node refs, nodeLinks). A call to nodeRelJoin is also done. Node join tries always to set a real conductor nA->mask (>= 0) and choices to use the lowest bottom coordinates for nA. When different, the two node groups are merged. The group notReady count is decremented for node nB.

In the most common case, nodeJoin is called by subnodeJoin. In that case, the nodes have always subnode refs. Function nodeJoin is also called in the readyGroup environment by reducMinSepRes. In that environment, the node group is ready (notReady = 0) and the nodes do not have subnode refs and polnode nodeLinks. In a special case, nodeJoin is called by makeContact (resEnumTile), to join distributed substrate contacts. In that case, there can be joined with a node, that does not have subnode refs anymore. The group notReady counter counts for the number of nodes that have subnode refs. Thus, when two nodes are joined, which have both subnode refs, the count is 2. And the count must be decremented to 1, because the result is one node with subnode refs. But, when two nodes are joined, which don’t have both subnode refs, the count is 1. Thus, in that case, the count may not be decremented. The code expects that node nB has subnode refs.

### 3.4 Function: nodeRelJoin

The function has two arguments, two node pointers (nA and nB). The node relations of nB are moved (added) to node nA. The following relations: (1) nodeTorLinks, (2) node "names" list, and (3) node net equivalences. Note that polnode relations must already been solved.

### 3.5 Function: readyNode

This function has a pointer to a node as argument. This node is ready, because (a) there is no subnode more pointing to it (node -> subs == NULL), or (b) there is no port more pointing to it (node -> ports == NULL). The group notReady count needs to be decremented for each ready node and port. When both conditions (a) and (b) are true, the node may be delayed and possible be eliminated. However, when the group becomes ready (notReady == 0), a call to readyGroup is done. Note that terminal nodes and nodes with a set keep flag are not delayed. Nodes with polnode links (node -> pols != NULL) are also not delayed.

## 4. NODE VARIABLES

### 4.1 Variable: inNod

Integer count, initialized to zero before each pass by initLump. Used for "nodes" extraction statistics (printed with option **-i**). Tells, howmany createNode calls are done.

### 4.2 Variable: outNod

Integer count, initialized to zero before each pass by initLump. Used for "nodes" extraction statistics (printed with option **-v** and **-i**). Tells, howmany outNode and outGndNode calls are done. Note, used for variable maxNodIfKept.

### 4.3 Variable: currIntNod

Integer count, initialized to zero before each pass by initLump. Used for "nodes" extraction statistics (printed with option **-i**). Is incremented for each call to createNode and decremented for each call to nodeDel. The end result should be zero. Function endLump gives a warning message (when the result is > 0), that there are circuit items left behind in core. Note, used for variable maxNodIfKept and maxIntANod.

### 4.4 Variable: rest_node_cnt

Function endLump prints between ’(’...’)’ in the "nodes" statistics the value of currIntNod (see above) on entrance of endLump minus 1. Note that endLump does always subnodeDel for the substrate node (see rest_grp_cnt). Thus, when the value of rest_node_cnt is not equal to the value of currIntNod, then there where left a number of nodes in the queue (unfinished groups).

### 4.5 Variable: maxIntNod

Integer count, initialized to zero before each pass by initLump. Used for "nodes" extraction statistics (printed with option **-i**). The maximum internal value of currIntNod (see above) in the pass.

### 4.6 Variable: condemnCnt

Integer count, initialized to zero before each pass by initLump. Used for "core nodes" extraction statistics (printed with option **-i**). Is incremented for each call to nodeDel and decremented for each call to disposeNode. The result must be zero, else assert failure. Note, used for variable maxIntANod.

### 4.7 Variable: maxIntANod

Integer count, initialized to zero before each pass by initLump. Used for "core nodes" extraction statistics (printed with option **-i**). The maximum internal value of currIntNod + condemnCnt (see above) in the pass.

### 4.8 Variable: maxNodIfKept

Integer count, initialized to zero before each pass by initLump. Used for "nodes" extraction statistics (printed with option **-i**). The maximum internal value of currIntNod + outNod (see above) in the pass.

### 4.9 Variable: eliNod

Integer count, initialized to zero before each pass by initLump. Used for "nbr. of nodes eliminated" extraction statistics (printed with option **-i**). Tells, howmany nodes are eliminated (function elim calls are done).

### 4.10 Variable: areaNodes

Integer count, initialized to zero before each pass by initLump. Used for "equi area nodes" extraction statistics (printed with option **-i**). Tells, howmany makeAreaNode calls are done.

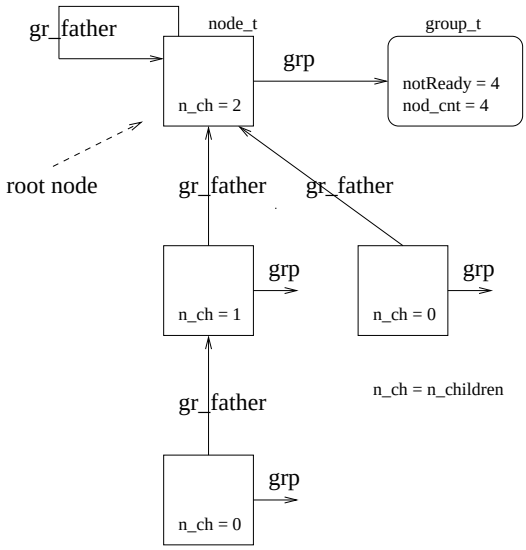
### 4.11 Variable: equiLines

Integer count, initialized to zero before each pass by initLump. Used for "equi line nodes" extraction statistics (printed with option **-i**). Tells, howmany makeLineNode calls are done.

### 4.12 Variable: areaNodesTotal

Integer count, initialized to zero before each pass by initLump. Used for "total ready area nodes" extraction statistics (printed with option **-i**). Tells, howmany area nodes are seen by function readyNode.

## 5. NODE GROUPS




When there are many nodes, an advantage of this data structure is, that only the root node need to be reconnected by a node join. It has as disadvantage, that condemned nodes only can be disposed when they don't have a child (see function findRoot). Note that the root node is always the last disposed node. Another disadvantage is, that not all nodes point to the group data. And the group data does not point to a node. Another disadvantage is, that all nodes of the group can not easy be found (they are not a double linked list). Nodes and group data are only created by function subnodeNew.

### 5.1 Grp

The macro Grp(node) returns a pointer to the group of the node. The Grp macro returns "node -> gr_father -> grp", if set, else "findRoot (node) -> grp" is returned. Note that Grp can return NULL when all the nodes are flagged as condemned. Note that Grp calls the findRoot function only for nodes, which path needs to be compressed.

### 5.2 findRoot

The findRoot function returns the root node of a node group. A node group has only one root node. When the root node is the last node of the group and this node is flagged as condemned, the root node is disposed and NULL is returned. The findRoot function compress also the node path to the root, it makes the root node the "gr_father" of the node. It does this compress process also for the node fathers of the node. It can also dispose the node, when the node is a leaf (n_children == 0) and is flagged as condemned. It can also dispose the fathers of the node, when the fathers become leaf nodes after disposal of the child node. Note that findRoot is the only function, that calls disposeNode function.

### 5.3 groupDel

Function groupDel disposes the group. It can also dispose the group "tileInfo" (when not equal NULL). It also decrements the currIntGrp counter. Normally, function mergeGrps or outGroup does groupDel.

### 5.4 mergeGrps

Two nodes need to be in the same group, when they are connected with each other by a conductor element of type 'G'. A root node is chosen, named node nB, the group data struct of this node group (grB) is deleted by function groupDel. The group data struct of node nA (grA) heritance all info from grB. Root node nB becomes a child node of root node nA (nB -> gr_father = nA; nB -> grp = NULL).

Function mergeGrps is called in the following functions:

1. nodeJoin When nodes nA and nB are joined (see nodeJoin, lump/lump.c), one node is deleted, but both groups are merged (when grA != grB).
2. elemNew When a conductor element of type 'G' is made between nodes nA and nB (see lump/elem.c), mergeGrps is called when grA != grB. This merge is not done for conductors of type 'S'.
3. elemAdd When a conductor element of type 'G' is made between nodes nA and nB (see lump/elem.c), and there is already a conductor element of type 'S' between both nodes, then this element type becomes 'G'. When both nodes are not in the same group, function mergeGrps is done.

### 5.5 Variable: outGrp

Integer count, initialized to zero before each pass by initLump. Used for "conductors" extraction statistics (printed with option **-i**). Tells, howmany outGroup calls are done. Note that outGroup does a call to groupDel. Variable "outGrp" is also used in prePass2 (see variable outPrePassGrp).

### 5.6 Variable: outPrePassGrp

Integer count, initialized to zero before each pass by initLump. Only used in prePass2, counts the selected node groups by outTileConnectivity. Note that outTileConnectivity is called by outGroup. The value is printed in verbose mode by endLump at the end of prePass2. Tells, that there "outPrePassGrp" (out of "outGrp") interconnects are selected for resistance extraction.

### 5.7 Variable: currIntGrp

Integer count, initialized to zero before each pass by initLump. Used for "conductors" extraction statistics (printed with option **-i**). Tells, howmany groups are allocated (by subnodeNew) and are remaining after dispose by groupDel (decrements currIntGrp). The result should be zero. Function endLump gives a warning message (when the result is >

0), that there are circuit items left behind in core.

### 5.8 Variable: rest_grp_cnt

Function endLump prints between ’(’...’)’ in the "conductors" statistics the value of currIntGrp (see above) on entrance of endLump minus 1. Note that endLump does always subnodeDel for the substrate node (subnSUB). This substrate group is not finished before. Function endLump does a "blockOutput" request, and readyGroup shall block the output of this node (does not outGroup, but sets "blockedNode"). Possibly there are nodes left in the queue, endLump shall do outGroup for these unfinished groups. After that the blocked substrate node is outputted, because it must be the last group. Note that the GND net is outputted after this substrate net, but the GND node does not have a group (subnGND == NULL). Thus, when the value of rest_grp_cnt is not equal to the value of currIntGrp, then there where a number of unfinished groups.

### 5.9 Variable: maxIntGrp

Integer count, initialized to zero before each pass by initLump. Used for "conductors" extraction statistics (printed with option **-i**). The maximum internal value of currIntGrp (see above) in the pass.

### 5.10 Variable: maxIntNodGrp

Integer count, initialized to zero before each pass by initLump. Used for extraction statistics (was printed with option **-i**). The message "max nod in g : ..." is currently not more printed.

## 6. NODE GROUP DATASTRUCT MEMBERS

Data that a group of nodes share together. This "group_t" data struct contains the following members (see "include/lump.h"):

```
typedef struct group {
    int notReady;           // ready group count
    int nod_cnt;            // node count
    int res_cnt;            // res elemNew count (for admin only)
    int cap_cnt;            // cap elemNew count (for admin only)
    unsigned supply : 8;    // flag for supply short detection
    unsigned prick  : 8;    // flag for selective res choice
    unsigned flag   : 8;    // flag for adjacent group detection
    int instNr;             // cell instance number
    struct groupTileInfo * tileInfo; // tile info for back-annotation
    struct Terminal * name; // net_name preference
} group_t;
```

### 6.1 Group Member: notReady

Init to 1 by subnodeNew. A node group becomes ready, when the group notReady counter becomes zero. In that case function readyGroup is called. Each time a node is added to the group (by mergeGrps), the notReady counter is incremented. Each time a node becomes ready (see readyNode), the notReady counter is decremented. However, the nodes can have links to transistor ports and BJT polnodes and other polnodes. These links also increments the group notReady counter. Thus, the node group becomes only ready, when all the links are done. Function nodeJoin shall also decrement the group notReady counter. The notReady counter is also adjusted for temporary nodes in the group. The notReady counter is also used to make a choice between the nodes in nodeJoin. And is used in readyGroup for adjacent group ready test.

### 6.2 Group Member: nod_cnt

Init to 1 by subnodeNew. The group node count is updated, when two node groups are merged by mergeGrps. The node count is decremented by nodeDel, when the node is condemned. Note that a node group can contain condemned nodes. The "nod_cnt" is used in mergeGrps, to delete the group with the less number of nodes. And "nod_cnt" is used by function elim, to delete the group (calling groupDel), when the "nod_cnt" is zero. Note that elim can possible delete the last node of a group. The "nod_cnt" is also used for statistics, to set maxIntNodGrp.

### 6.3 Group Member: res_cnt

Init to 0 by subnodeNew. It counts the number of resistance elements in the node group. This is only for statistics, to set maxIntResGrp.

### 6.4 Group Member: cap_cnt

Init to 0 by subnodeNew. It counts the number of capacitance elements in the node group. This is only for statistics, to set maxIntCapGrp.

### 6.5 Group Member: supply

Init to 0 by subnodeNew. Function nameAdd sets the supply flag, when the label or terminal name is the name of the positive supply ("vdd") or negative supply ("vss" or "gnd"). The supply flag is 1 for negative supply and 2 for positive supply. When a short between both supplies is detected by function nameAdd or mergeGrps a warning message is given and the flag is set to 3. Thus, the supply flag is only for short checking.

### 6.6 Group Member: prick

Init to 0 by subnodeNew. This flag is only used in prePass2 for selective resistance extraction. The node group is selected, when the prick flag is set. The prick flag is set in testTileXY (called by scan, the edges must have tiles with inited subnodes), when the coordinate point is in the tile. The prick flag is also set in nameAdd, when a terminal or label name is equal the prickName.

### 6.7 Group Member: flag

Init to 0 by subnodeNew. This flag is only used by function readyGroup. The flag is set for an adjacent group, when a node of that group is put in the adjacent groups queue (QAG). Thus, only one node of each different adjacent group can be placed in the QAG queue. Note, a ready group can only be outputted by outGroup, when all its adjacent groups are ready. Note that, the ready adjacent groups are also checked for all adjacent ready. The adjacent group flags are reset at the end of readyGroup.

### 6.8 Group Member: instNr

Initialized outside subnodeNew. The "instNr" is only set and used, when optReadNetTerm (option -%M) and optNoCore (option -%K) are TRUE. The node group gets its "instNr" from the tile "instNr". Each tile gets its "instNr" by createTile, which calls function findInstNr. Function findInstNr searches for the "instNr" in file "3Ddecom". Note that no subnodeJoin/Copy are done for unequal tile "instNr". The group "instNr" is also written as net attribute by outNode.

### 6.9 Group Member: tileInfo

Init to NULL by subnodeNew. However, by prePass2 or optBackInfo or optNetInfo it points to a new created "groupTileInfo_t" data struct. In prePass2 (selective resistance extraction, options **-j** and **-k**), the selected high resistive tile parts (geometry information) is here stored, and written into a file (which is used in the extract pass). By optBackInfo (options **-x** and **-t**), back-annotation info is stored in the extrPass into this data struct. By optNetInfo (option **-y**, program model OPTEM), special back-annotation info is also stored in the extrPass into this data struct.

### 6.10 Group Member: name

Init to NULL by subnodeNew. For "struct Terminal" (typedef "terminal_t") see "include/terminal.h". The group name is used in the extrPass as net name (in functions

outGroup and outNode). But not for the name of the substrate node net. The group names are not used, when makeMaskXYName (parameter "node_pos_name") is set. A group name can only be a terminal name (of the extracted root cell) when parameter "term_is_netname" is set. Else, group names are label names. When there is a choice between different names (see groupNameAdd and mergeGrps), type tLabel2 names have lower priority then type tLabel and tTerminal names. For names with equal priority, the name with the lowest coordinates is chosen.

## 7. NODE DATASTRUCT MEMBERS

The "node_t" data struct contains the following members (see "include/node.h"):

```
typedef struct Node {
    unsigned substr  : 4;
    unsigned term    : 4;
    unsigned area    : 4;
    unsigned flag    : 4;
    unsigned flag2   : 4;
    unsigned flag3   : 4;
    unsigned keep    : 4;
    unsigned delayed : 4;
    struct Terminal * names;
    struct subnode  * subs;
    struct element ** con;
    struct element ** cap;
    int res_cnt;
    int cap_cnt;
    int n_n_cnt;  /* names and netEq count */
    double * gndCap;
    double * substrCap;
    double * substrCon;
    double * moments;
    double weight;
    double help;
    struct nodeLink    * pols;
    struct nodeTorLink * ports;
    struct netEquiv * netEq;
    struct group * grp;
    struct Node  * gr_father;
    struct Node  * next, * prev;
    int    condemned;
    int    n_children;
    int    mask;
    coor_t node_x, node_y;
    int id;
    int degree;
} node_t;
```

### 7.1 Node Member: ports

A pointer to "nodeTorLink_t". The node is also a port of a transistor. We name the "nodeTorLink_t" data struct a "port". Note that port->n points to the node and port->t points to the "transistor_t" data struct. And port->nextN points to the next "port" of the node.

### 7.2 Node Random Id's

See "lump/node.c". The node random id generator is now initialized for each space pass. Added function srand48 to function nqInit. Thus the nodes have always the same id's in each pass and space shall always choice the same node for certain node eliminations (in the extract pass).

### 7.3 Old Element Hashing Code

See "lump/elem.c". The ELEMHASH macro calculates a hash value based on the difference of the node id's. Therefor node nB->id must be greater (or equal) node nA->id. I have removed the old ELEMHASH code, which was also using a Sort array. I have also removed the calls of function enlargeElemHT from function returnElement, because it can better only be done in function elemNew.

## 8. NODE EQUIVALENCES

Each outputted node (function outNode) can have a linked list of net equivalences. And each outputted node can create net equivalences to other nodes (done by outResistor, outCapacitor and outAreaPerimElement). Note that these nodes can be in adjacent groups. The Node, that creates the net equivalence by the other node, shall also remove the element between the two nodes. Only the first processed node shall make the equivalence, and the second processed node shall use it. Node equivalences are also made by outVBJT, outLBJT and outTransistor. Note that all node equivalences must be done before the node is done by outNode. Note that outNode is called by outGroup, and that outGroup is only done, when all adjacent groups are ready. Note that the nets of cell instances are made with the node "names" member, which is a linked list of terminal and label names.

### 8.1 Node Equivalence Data Struct

The node net equivalence data struct contains the following members (see "include/lump.h"):

```
typedef struct netEquiv {
    char  instType;
    char  term;
    int   number;
    char *instName;     /* if defined: overrules number */
    struct Node * net;
    struct netEquiv * nextDsRing;
    struct netEquiv * next;
} netEquiv_t;
```

### 8.2 Function: addNetEq

This function creates net equivalence data for a node.

### 8.3 Function: addDsNetEq

Variant of function addNetEq, which is called by outTransistor, when optDsConJoin is TRUE. There is made a dsRing of ’d’ ports. Note that unconnected net equivalence ports can be removed by outNode.

### 8.4 Variable: optDsConJoin

Boolean, this option is default TRUE. Default, there are no separate d/s boundaries for transistors. Users, who want to experiment, can use the special option **-J**, to set it FALSE.

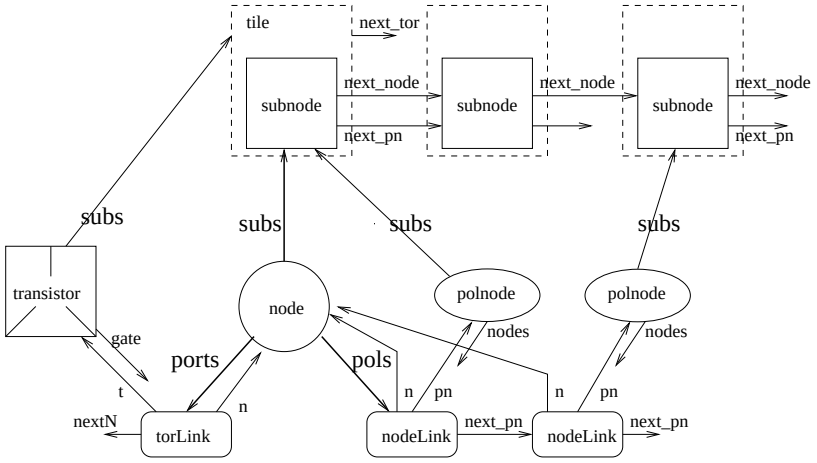
### 8.5 Variable: currNeqv

Integer count, initialized to zero before each pass by initLump. Counts the netEquiv_t data structs, that are allocated by addNetEq/addDsNetEq. And the count is decremented for each netEquiv_t, that is disposed by outNode. The end result should be zero. Function endLump gives a warning message (when the result is > 0), that there are

"currNeqv" net equivalences left behind in core.

## 9. NODE LINKS

A node has a pointer to a list of subnodes, but also pointers to a list of nodeLinks (node/polnode links) and to a list of torLinks (node/tor links). See the figure below.




Note that in function nodeJoin the node links must be updated.

### 9.1 Node Tor Links

The data structs of a nodeTorLink and transistor (tor) are given below:

```
typedef struct nodeTorLink { // "include/lump.h"
    char type;  /* 'b', 'g' or 'd' */
    struct Node       * n;
    struct transistor * t;
    struct nodeTorLink * prevN, * nextN;
    struct nodeTorLink * prevT, * nextT;
} nodeTorLink_t;

typedef struct transistor { // "include/transist.h"
    struct elemDef * type;
    ...
    struct nodeTorLink * gate, * drains, * bulk;
    struct Tile * subs;
    struct dsBoundary * boundaries;
} transistor_t;
```

The "nextN" points for "ports" of a node to the next nodeTorLink (another tor port) for that node (ports->n == ports->nextN->n). The "nextT" points for "drains" of a transistor to the next type 'd' nodeTorLink (another node) for that transistor (drains->t == drains->nextT->t). Note that transistors have only one "gate" and one (optional) "bulk" nodeTorLink (see outTransistor). Note that the node group notReady counter is incremented for each reference to a node (see nTLinkAdd/New) and is decremented for each node reference that is removed (see nTLinkDel). Thus, a node group can only become ready, when all these node references of nodes in the group are solved.

### 9.2 Node Polnode Links

Data struct "nodeLink_t" (see "include/polnode.h").

```
typedef struct nodeLink {
    struct Node     * n;
    struct polnode  * pn;
    struct nodeLink * next_n;
    struct nodeLink * next_pn;
} nodeLink_t;
```

The "next_pn" points for "pols" of a node to the next nodeLink (another polnode) for that node (pols->n == pols->next_pn->n). The "next_n" points for "nodes" of a polnode to the next nodeLink (another node) for that polnode (nodes->pn == nodes->next_n->pn). The chosen notation looks nice, but is not identical to the chosen notation for node tor links and node subnode references. Note that the node group notReady counter is incremented for each added nodeLink (by function nodeLinkAdd) and decremented for each removed nodeLink (by function nodeLinkDel). Nodes can have different polnodes and polnodes can point to different group nodes (see nodepoints). Thus, nodes and polnodes, which have nodeLinks, belong to the same node group. A node group can only be ready, when the nodeLinks (and polnodes) are done. See function readyNode, a node with pols is not delayed and the group notReady must be > 0.

## 10. NODE TOR LINK FUNCTIONS

### 10.1 Function: nTLinkAdd

Only called by functions nodeRelJoin, subtorJoin and portAdd. The function has three arguments: (1) a port type (’g’, ’d’ or ’b’), (2) a node pointer, and (3) a transistor pointer. The transistor may not have two links of the same port type to the same node. However, different port types may point to the same node. It calls nTLinkNew for a new nodeTorLink.

### 10.2 Function: nTLinkNew

Only called by function nTLinkAdd to create a new nodeTorLink between node and transistor. The node group notReady counter is incremented.

### 10.3 Function: nTLinkDel

Only called by functions nodeRelJoin, subtorJoin, portAdd and outTransistor. The function has one argument, the nodeTorLink pointer. It calls readyNode to decrement the node group notReady counter. The node is not delayed, because its "term" flag is set. Function readyGroup is called, when notReady becomes zero. It removes the nodeTorLink references from the node "ports" list and from the transistor port type list, and disposes the nodeTorLink.

### 10.4 Function: subtorJoin

The function has two arguments, two tile pointers. The two tiles are gate areas of the same transistor. The tor tB of tileB is joined with the tor tA of tileA and tB is disposed. The nodeTorLinks of tB are added to tA (when needed).

### 10.5 Function: portAdd

The function has three arguments, (1) a subnode pointer, (2) a tile pointer, and (3) a port type (’g’, ’d’ or ’b’). The ’d’ ports, all nodeTorLinks are made. For ’g’ and ’b’ ports, there may only be one nodeTorLink. When there are two different nodes, nTLinkDel is called for the old nodeTorLink.

### 10.6 Function: outTransistor

The function has one argument, the transistor pointer. It removes the nodeTorLink references. In the last pass, it calls addNetEq to create node net equivalences and it puts a transistor instance record to stream CIR_MC. At last, the transistor is disposed by torDel. Note that function outTransistor is only called by subtorDel (by clearTile), when the last (gate) tile of the transistor is done.

### 10.7 Node BJT Links

A BJT device points with a node array to nodes. For each node that is pointed to, the node group notReady counter is incremented in function newVBJT/LBJT. For each device that is deleted, the notReady counter is decremented in function deviceDel. Note

that a node itself does not directly point to a BJT link, that does the polnode (see polnode links).

## 11. POLNODES

The polnodes are only used, when there are bipolar elements in the technology file and therefor mode "hasBipoElem" is set. When hasBipoElem is set, enumPair shall call bipoPair and enumTile shall call bipoTile for handling these possible existing bipolar elements.

Note that in hasBipoElem mode, a subnode must have a "pn" member. After a call to subnodeNew there must be done a call to polnodeAdd or polnodeCopy. However, after subnodeCopy, subnodeDel or subnodeJoin needs nothing to be done.

Note that some subnodes do not have polnodes, for example the subnodes of distributed substrate contacts, and temporary used subnodes.

The "polnode_t" data struct contains the following members (see "include/polnode.h"):

```
typedef struct polnode {
    char type; /* 'n' (electrons), 'p' (holes) or 'a' (metal) */
    int sortNr; /* resistance-type */
    coor_t xl, xr, yb, yt;
    struct pnEdge    * edges;
    struct subnode   * subs;
    struct junction  * j;
    struct pnTorLink * t;
    struct nodeLink  * nodes;
} polnode_t;
```

### 11.1 Polnode Data Struct Members

### 11.2 Function: polnodeAdd

This function has 3 arguments (subnode, conductor nr and type). This function needs to be called after subnodeNew (this is done in enumPair). The function creates a new polnode for the subnode and initialize the polnode data. It makes a nodeLink between the node and the polnode. Just in case, when the subnode already has a polnode, the function does nothing.

### 11.3 Function: polnodeCopy

This function has two subnode arguments (snA and snB). The function does not create a new polnode for subnode snB, but lets snB points to the polnode of subnode snA. When snB already has a polnode, the function does nothing. Subnode snB becomes the head of the subnode linked list of the polnode. Subnode snA is also somewhere in this linked list. When both subnodes point to different nodes, the node of snB gets a nodeLink to the polnode (by nodeLinkAdd).

### 11.4 Function: polnodeJoin

This function has two polnode arguments (pnA and pnB). The two polnodes are joined, pnA is updated with data from pnB, and polnode pnB is disposed. Note that pnA is updated with BJT, junction and edge data of pnB. The nodeLinks of pnB are added to

pnA.

There are two functions, which can call the polnodeJoin function, subnodeJoin and mkConnect. Function mkConnect joins subnodes for bipolar connect elements, these subnodes must have polnodes with the same carrier-type. By resistance extraction and when the connect element has resistance and area and quasi3D is TRUE, then mkConnect adds a conductor element between the two subnodes and can call polnodeJoin for the polnodes. For all cases, the following polnode conditions must be true:
(a) only for two different polnodes,
(b) only for equal conductor carrier-type "type" (’a’, ’n’ or ’p’),
(c) only for equal "sortNr" (conductor nr) (#ifdef LUMPED_MODEL) (see "include/config.h", default defined).

### 11.5 Function: polnodeDel

This function has one polnode arguments (pn). The function is only called by subnodeDel, for the last subnode reference (!pn -> subs) of the polnode. The references of the polnode are checked. Ready BJT references are outputted and deleted (deviceDel), and ready junctions also (junctionDel). The polnode is disposed, when its junction and BJT references are gone.

### 11.6 Function: polnodeDispose

This function has one polnode arguments (pn). The function disposes the polnode. The pointers to subnode, junction and BJT data must already be NULL. Existing pnEdges are removed by pnEdgeDel, and existing nodeLinks are removed by nodeLinkDel. This function is called by polnodeJoin and can be called by polnodeDel, junctionDel and deviceDel.

### 11.7 Variable: hasBipoElem

Variable hasBipoElem is set, when the technology file contains one or more bipolar elements. These bipolar elements are VBJTELEM, LBJTELEM and PNCONELEM (see "extract/gettech.c"). Note that there is also a parameter "do_bipo", which can set or unset hasBipoElem. The unset can be important for layouts, which do not have bipolar elements, while using a technology file with bipolar elements.

### 11.8 Variable: hasBipoSub

Variable hasBipoSub is set, when one or more BJT elements in the technology file has a substrate connection. This variable is only used by initLump, to give the substrate node a polnode.

### 11.9 Variable: inPolnode

Integer count, initialized to zero before each pass by initLump. Used for extraction statistics (printed with option **-i**). Tells, howmany polnode_t are allocated (polnodeAdd’s are done).

### 11.10 Variable: outPolnode

Integer count, initialized to zero before each pass by initLump. Used for extraction statistics (printed with option **-i**). Tells, howmany polnodeDel calls are done.

### 11.11 Variable: currIntPolnode

Integer count, initialized to zero before each pass by initLump. Used for extraction statistics (printed with option **-i**). Tells, howmany polnode_t are allocated (see inPolnode) and are remaining after polnodeDispose (decrements currIntPolnode). The result should be zero. Function endLump gives a warning message (when the result is > 0), that there are bipolar circuit items left behind in core.

## 12. POLNODE LINKS

### 12.1 Polnode Node Links

### 12.2 Polnode BJT Links

Data struct "pnTorLink_t" (see "include/polnode.h").

```
typedef struct pnTorLink {
    int type;
    struct polnode * pn;
    union { struct vBJT * vT; struct lBJT * lT; } t;
    struct pnTorLink * next;
} pnTorLink_t;

typedef struct vBJT {
    struct elemDef * type;
    ...
    struct Node    * n[4]; // CO,EM,BA,SU
    struct polnode * pn[4];
} vBJT_t;
```

### 12.3 Polnode Junction Links

```
typedef struct junction {
    int sidewall;
    struct polnode  * pn[2]; // AN,CA: anode and cathode region
    struct junction * next[2]; // anode and cathode next
} junction_t;
```