SPACE CAP/RES ELEMENT FUNCTIONS APPLICATION NOTE

S. de Graaf

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

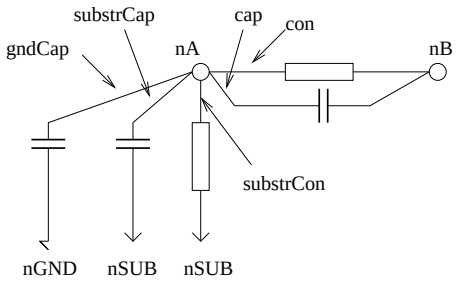
Report EWI-ENS 04-08
August 24, 2004

Last revision: September 2, 2004.

1. INTRODUCTION

The *space* program uses a number of functions to add, change, find and delete capacitors and resistors to nodes. This application note explains how these functions work.

The nodes points to these capacitors and resistors. Other pointers are used for the elements to ground and substrate nodes. See the code fragment of the node_t data structure in appendix B.



The values of capacitors to ground/substrate and resistors to substrate are stored in a double array. Note that there do not exist resistors to ground. Only between two real nodes are element_t data structures used. See for the element_t data struct appendix C.

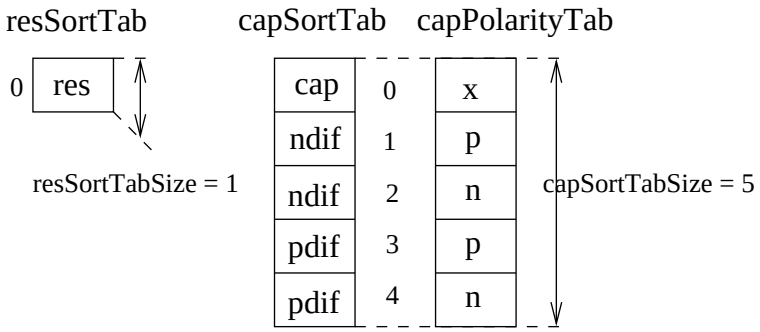
Each node is initialized to zero by function createNode. This function gives each node also an unique node id, which is used by the element hash function. Note that each node represents a conducting layout connection, which has a low resistivity value. This low resistivity value for *space* can be chosen with parameter "low_sheet_res".

The node cap_cnt and res_cnt counts only the number of element_t connected to that node. These counts are incremented / decremented by function nqChange. This function shall also calculate the node degree, which depends on the cap_cnt and res_cnt value. Besides that, the node cap_cnt / res_cnt are only used to choice which node must be deleted in function nodeJoin. Note that there are also cap_cnt / res_cnt members in the group_t data struct.

1.1 Element Sorts

The length of the element cap / con array depends on capSortTabSize / resSortTabSize. Note that also the double arrays depends on these sizes. The sizes are always ≥ 1 (the default size is 1). Note that the technology file is used to define different sorts. See the "Space User's Manual" section 3.4.8 "The conductor list", section 3.4.12 "The contact list" and section 3.4.13 "The capacitance list". The default sorts, installed on entry 0 in the capSortTab / resSortTab, have the type names "cap" / "res". Note that in the manual the keyword "type" is used, but in the *space* program, we use the term "sort". Note that other sorts seldom are used for conductors, thus resSortTabSize is almost always 1. But for capacitances it is more common and the sorts with different polarity (i.e. junction capacitances) use 2 entries. Thus, for capacitances, there is also a capPolarityTab. Note that entry 0 does not have a polarity (char 'x' is used in the table). The positive polarity

(char 'p') is always first specified in the table and one entry higher the negative (char 'n') polarity.
See for example the following tables:



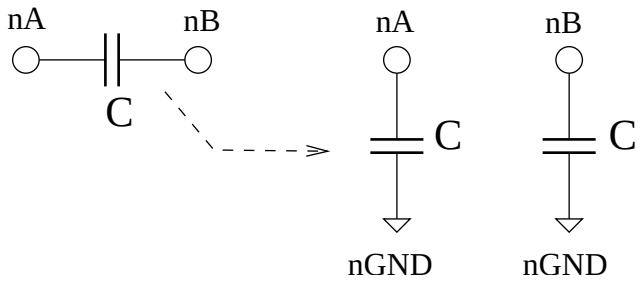
Note that each resElemDef_t, contElemDef_t and capElemDef_t has a sortNr member. This sortNr specifies for capacitance elements the first entry (positive entry) in the table. Note that function initLump can change the capSortTab and capSortTabSize, when there are junction capacitance entries. Note that entry 0 (sort number 0) is always used for 3D caps and 2D/3D substrate resistors.

1.2 Element Types

The *space* program functions use 3 types, type 'C' for cap elements, type 'G' and 'S' for res elements. Type 'S' is a special type 'G' and is used for substrate conductor elements. The nodes of type 'G' elements are all placed in the same conductor group (see elemAdd / elemNew). This is not done for conductor elements of type 'S'.

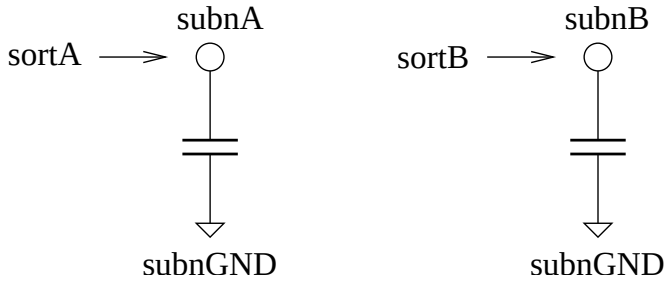
1.3 Function elemAdd

Function capAdd is most times used to call elemAdd('C',...). Note that function momentsElimOrWeight directly calls elemAdd. Note that capAdd redirects the cap value to array gndCap or substrCap when one of the subnodes is subnGND or subnSUB. Note that both subnodes may not be subnGND or subnSUB or subnGND and subnSUB. Note that subnA $\neq$ 0 means, that subnA != subnGND is.
Note that couple caps are redirected to GND when optCoupCap $\equiv$ false. The value, however, is not divided by two (see picture).



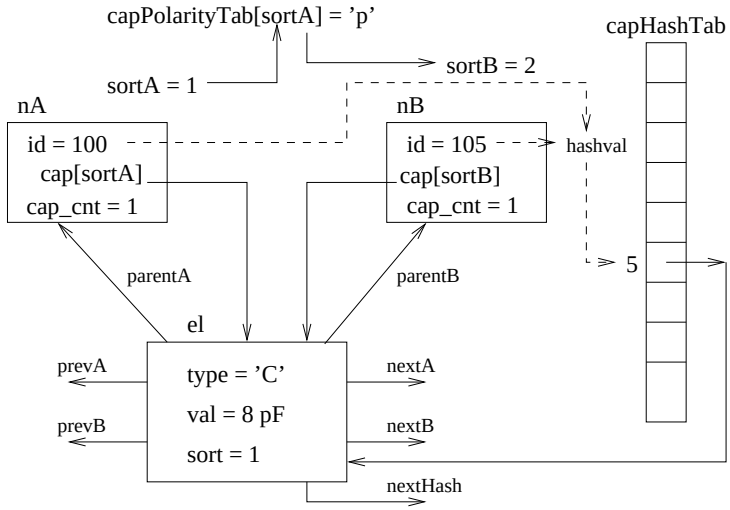
Note that argument "sort" is always used for subnA (sortA). Thus, when subnA $\equiv$

subnGND the cap value is assigned to sortB. The value of sortB depends on the polarity of sortA. When the polarity of sortA is 'p', the value of sortB is sortA+1. Note that for junction capacitances default the polarity is not used. The capSortEnable array shall redirect all given 'sort' values to sort 0. Parameter "jun_caps" must be specified to use the polarity method. Except for drain/source capacitances, they use always the polarity method. Note that the type names of d/s caps are prefixed with a '$' sign.
Note that elemAdd also calculates the value for sortB. Note that resistor elements don't have polarity, thus sortA ≡ sortB.



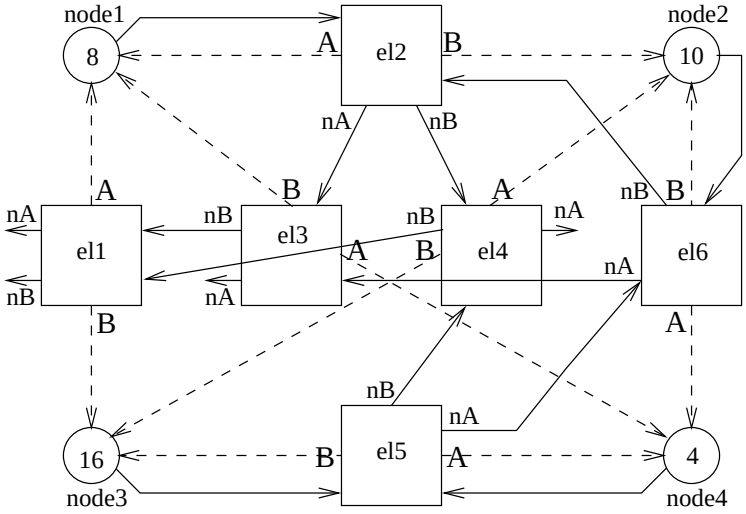
Function conAdd is most times used to call elemAdd('G',...). Note that the value of resSortEnable is always true. No different sort values are used (sortA ≡ sortB). When one of the subnodes is subnSUB, then the value is redirected to array substrCon and elemAdd is not called. This happens for contacts between interconnect and substrate.

After elemAdd / elemNew the following connections between the nodes and the element exist:



Note that function elemNew is always called with nA→id <= nB→id. When nA→id > nB→id both nodes are swapped and sortA becomes 2. In that case el→sort becomes 2. Thus, the el→sort value is always belonging to el→parentA. The node el→parentA has always a lower (or equal) node id as node el→parentB. The node id's are used for the hashvalue and are random chosen to get a better hashing.

Note that between two nodes only one cap / con element with the same sort can be found. The previous pointers (prevA / prevB) are only used for faster element delete from node list. The next element for a node can be found with macro NEXT, nextA is the next element when node is the parentA of current element. There is also a PREV macro (see appendix D). Note that macro OTHER returns the other node for a element. Note that function elemDel uses el→nextA to put the deleted element into elemList, and function elemNew shall try to get a new element from elemList. The following figure shows all possible elements (for one sort) for 4 nodes (6 elements).



1.4 Function momentsElimOrWeight

Because function momentsElimOrWeight directly calls elemAdd, function elemAdd must test the nodes for this special situation. Because it is possible that one of the nodes is nGND (≡ 0) or nSUB.

Note that momentsElimOrWeight must call elemAdd directly, because only elemAdd can add moments to cap elements or to nodes (for ground caps).

Function momentsElimOrWeight can be called by function elim or updateWeight. Function updateWeight is called by nqDelayElim, which is called by readyNode / readyGroup. But there can only be moments when optIntRes and optCap is true, and option **-G** or **-g** is used with the *space* program.

1.5 Global Variable dontMakeNegVal

The variable dontMakeNegVal is only set in file "extract/latcap.c", when function updateLateralCap is called (using option **-l**). Function updateLateralCap calls function compensateEdgeCap, to compensate the value of existing edge capacitors, and there dontMakeNegVal is set to true. Only just in case (although it is unlikely to occur in practice), to make sure that we do not get negative capacitance in the network. The edge caps can be couple caps to subnGND.

2. APPENDICES

APPENDIX A -- Recent Changes of Source Files

```
================================================================================
File: elem.c            New revision:   4.44    Tue Aug 24 12:19:38 2004
Comments:
changed ELEMHASH and returnElement (SdeG)
================================================================================
File: elem.c            New revision:   4.45    Tue Aug 24 16:02:42 2004
Comments:
removed sort arg. from elemDel; changed next into el_next; sort into sortA;
added Swap
================================================================================
File: elim.c            New revision:   4.32    Tue Aug 24 16:02:43 2004
Comments:
removed sort arg. from elemDel; changed next into el_next
================================================================================
File: extern.h          New revision:   4.46    Tue Aug 24 16:02:45 2004
Comments:
removed sort arg. from elemDel; changed sort into sortA
================================================================================
File: lump.c            New revision:   4.98    Tue Aug 24 16:02:46 2004
Comments:
removed sort arg. from elemDel; changed next into el_next
================================================================================
File: out.c             New revision:   4.99    Tue Aug 24 16:02:47 2004
Comments:
removed sort arg. from elemDel; changed next into el_next
================================================================================
File: reduc.c           New revision:   4.47    Tue Aug 24 16:02:48 2004
Comments:
removed sort arg. from elemDel
================================================================================
```

APPENDIX B -- Data Struct node_t

```
// see: include/node.h

typedef struct Node {

    int id; // The unique id number of the node.

    int degree;

    int res_cnt; // Number of resistances  connected to this node.
    int cap_cnt; // Number of capacitances connected to this node.

    /* The conducting elements connected to this node.
     *
     * This field is a pointer to an array of resSortTabSize element pointers.
     * The pointer may be NULL when the array is considered "not available".
     * The array is indexed by "sort" (see resSortTab).
     */
    struct element ** con;

    /* Total admittance to the substrate node.
     * This field contains a pointer to an array of resSortTabSize values.
     */
    double * substrCon;

    /* The capacitive elements connected to this node.
     *
     * This field is a pointer to an array of capSortTabSize element pointers.
     * The pointer may be NULL when the array is considered "not available".
     * The array is indexed by "sort" (see capSortTab).
     */
    struct element ** cap;
    double * gndCap;
    double * substrCap;

    /* The extra moments to ground of this node.
     */
    double *moments;

} node_t;
```

APPENDIX C -- Data Struct element_t

```
// see: include/lump.h

typedef struct element {
    struct element *prevA, *nextA;
    struct Node * parentA;
    struct element *prevB, *nextB;
    struct Node * parentB;
    struct element * nextHash;
    double val;
    double *moments;
    char type;
    char flag;
    short sort;   /* sort of element for parentA */
} element_t;
```

APPENDIX D -- Element Functions Code

```
// see: lump/define.h

#define PREV(el,node)  (el->parentA == node ? (el)->prevA : (el)->prevB)
#define NEXT(el,node)  (el->parentA == node ? (el)->nextA : (el)->nextB)
#define APREV(el,node) (el->parentA == node ? (&(el)->prevA) : (&(el)->prevB))
#define ANEXT(el,node) (el->parentA == node ? (&(el)->nextA) : (&(el)->nextB))
#define OTHER(el,node) (el->parentA == node ? (el)->parentB : (el)->parentA)

// see: lump/elem.c

#define ELEMHASH(n1,n2,size)  ((n2 -> id - n1 -> id) % size)

bool_t dontMakeNegVal = FALSE;

static element_t * elemList = NULL;
static element_t ** capHashTab;
static element_t ** resHashTab;

static int capHT_size = 0;
static int resHT_size = 0;

Private void elemNew (type, nA, nB, val, sortA, sortB, moments)
int type;
node_t *nA, *nB;
double val;
int sortA, sortB;
double * moments;
{
    element_t *el, *el_nextA, *el_nextB;
    int hashval;

    if (!elemList)   el = NEW (element_t, 1);
    else elemList = (el = elemList) -> nextA;

    el -> val = val;
    el -> type = type;
    el -> flag = 0;
    el -> sort = sortA;
    el -> parentA = nA;
    el -> parentB = nB;
    el -> prevA = NULL;
    el -> prevB = NULL;
    el -> moments = moments ? newMoments (moments) : NULL;

    if (type == 'C') {
        el_nextA = nA -> cap[sortA];
        el_nextB = nB -> cap[sortB];
        nA -> cap[sortA] = el;
        nB -> cap[sortB] = el;
    }
    else {
        el_nextA = nA -> con[sortA];
        el_nextB = nB -> con[sortB];
        nA -> con[sortA] = el;
        nB -> con[sortB] = el;
    }
    el -> nextA = el_nextA;
    el -> nextB = el_nextB;
    if (el_nextA) *APREV (el_nextA, nA) = el;
    if (el_nextB) *APREV (el_nextB, nB) = el;
```

```
    if (type == 'C') {
        if (++currIntCap > maxIntCap) maxIntCap = currIntCap;
        if (currIntCap + outCap > maxCapIfKept) maxCapIfKept = currIntCap + outCap;
        nqChange (nA, 0, 1);
        nqChange (nB, 0, 1);
        ...
        if (2 * currIntCap > capHT_size) enlargeElemHT (&capHashTab, &capHT_size);
        hashval = ELEMHASH (nA, nB, capHT_size);
        el -> nextHash = capHashTab[hashval];
        capHashTab[hashval] = el;
    }
    else {
        if (++currIntRes > maxIntRes) maxIntRes = currIntRes;
        if (currIntRes + outRes > maxResIfKept) maxResIfKept = currIntRes + outRes;
        nqChange (nA, 1, 0);
        nqChange (nB, 1, 0);
        if (Grp(nB) != Grp(nA)) {
            if (type == 'G') { mergeGrps (nA, nB); ... }
        }
        else {
            el -> type = 'G'; ...
        }
        if (2 * currIntRes > resHT_size) enlargeElemHT (&resHashTab, &resHT_size);
        hashval = ELEMHASH (nA, nB, resHT_size);
        el -> nextHash = resHashTab[hashval];
        resHashTab[hashval] = el;
    }
}

Private void enlargeElemHT (elemHT, HT_size)
element_t ***elemHT;
int *HT_size;
{
    element_t *el, *el_next, **newHT, **oldHT;
    int i, new_size, old_size, hashval;

    old_size = *HT_size;
    new_size = (old_size == 0)? 500 : old_size * 2;
    oldHT = *elemHT;
    *elemHT = newHT = NEW (element_t *, new_size);
    *HT_size = new_size;

    /* initialize new hash table */
    for (i = 0; i < new_size; i++) newHT[i] = NULL;

    /* move elements from old hash table to new hash table */
    for (i = 0; i < old_size; i++) {
        el = oldHT[i];
        while (el) {
            el_next = el -> nextHash;
            hashval = ELEMHASH (el -> parentA, el -> parentB, new_size);
            el -> nextHash = newHT[hashval];
            newHT[hashval] = el;
            el = el_next;
        }
    }

    /* delete old hash table */
    if (old_size > 0) DISPOSE (oldHT);
}
```

```
/*! @brief Add a circuit element between two nodes.

    This function adds a circuit element between the two nodes @c nA and
    @c nB. The type of circuit element is specified by the parameter @type,
    and can be 'C' (capacitor), 'G' (conductor) or 'S' (substrate conductor).

    The @c sort parameter designates the sort of the element at node A. Note
    that if the element is a junction capacitance (a fact which can be derived
    from the @sort parameter), then @c nA and @c nB will have different
    polarities, and the algorithm will automatically detect this case.
*/

void elemAdd (type, nA, nB, val, sortA, moments)
int type;
node_t *nA, *nB;
double val;
int sortA;
double *moments;
{
    element_t * el;
    int sortB;

    if (nA == nB) return;

    sortB = sortA;
    if (type == 'C') {
        if (capPolarityTab[sortA] == 'p') ++sortB;
        else if (capPolarityTab[sortA] == 'n') --sortB;

        if (!nA) { // #ifdef -------- MOMENTS --------
            addGndCap (nB, val, sortB, moments);
            return;
        }
        else if (!nB) {
            addGndCap (nA, val, sortA, moments);
            return;
        }
        else if (!optCoupCap) {
            addGndCap (nB, val, sortB, moments);
            addGndCap (nA, val, sortA, moments);
            return;
        } // #endif -------- MOMENTS --------

        if (nA == nSUB || nB == nSUB) {
            if (nB == nSUB) {
                nB = nA;
                sortB = sortA;
            }
            nB -> substrCap[sortB] += val;
            if (moments) { // #ifdef -------- MOMENTS --------
                if (!nB -> moments) nB -> moments = newMoments (moments);
                else { int i;
                    for (i = 0; i < extraMoments; i++)
                        nB -> moments[i] += moments[i];
                }
            } // #endif -------- MOMENTS --------
            return;
        }
    }
    else {
        ASSERT (type == 'G' || type == 'S');
        ASSERT (!moments);
        ASSERT (nA != nSUB && nB != nSUB); /* must be in substrCon[] */
    }
```

```
    ASSERT (nA && nB);

    if (nA -> id > nB -> id) {
        Swap (node_t *, nA, nB);
        Swap (int, sortA, sortB);
    }

    if ((el = returnElement (type, nA, nB, sortA))) {

        if (dontMakeNegVal && el -> val >= 0.0 && el -> val + val <= 0.0) {
            elemDel (el);
            return;
        }
        if (type == 'G' && el -> type == 'S') {
            el -> type = 'G';
            if (Grp(nB) != Grp(nA)) { mergeGrps (nA, nB); ... }
            ...
        }

        el -> val += val;

        if (moments) { // #ifdef -------- MOMENTS --------
            if (!el -> moments) el -> moments = newMoments (moments);
            else { int i;
                for (i = 0; i < extraMoments; i++)
                    el -> moments[i] += moments[i];
            }
        } // #endif -------- MOMENTS --------
    }
    else { /* not present yet between nA and nB */

        if (dontMakeNegVal && val <= 0.0) return;

        elemNew (type, nA, nB, val, sortA, sortB, moments);
    }
}

element_t * findElement (type, nA, nB, sortA)
int type; node_t *nA, *nB; int sortA;
{
    ASSERT (nA && nB);

    if (nA -> id > nB -> id) {
        Swap (node_t *, nA, nB);
        if (type == 'C') {
            if (capPolarityTab[sortA] == 'p') ++sortA;
            else if (capPolarityTab[sortA] == 'n') --sortA;
        }
    }
    return (returnElement (type, nA, nB, sortA));
}

Private element_t * returnElement (type, nA, nB, sortA)
int type; node_t *nA, *nB; int sortA;
{
    int hashval; element_t *el;

    if (type == 'C') {
        if (capHT_size == 0) return (NULL);
        hashval = ELEMHASH (nA, nB, capHT_size);
        el = capHashTab[hashval];
    }
    else {
        if (resHT_size == 0) return (NULL);
```

```
        hashval = ELEMHASH (nA, nB, resHT_size);
        el = resHashTab[hashval];
    }

    for (; el; el = el -> nextHash) {
        if (el -> parentA == nA && el -> parentB == nB
                                && el -> sort == sortA) return (el);
    }
    return (el);
}

void elemDel (el) element_t *el;
{
    element_t *e_p, *e_n;
    int hashval, sortA, sortB;
    node_t  *paA, *paB;

    sortA = sortB = el -> sort;
    if (el -> type == 'C') {
        if (capPolarityTab[sortA] == 'p') ++sortB;
        else if (capPolarityTab[sortA] == 'n') --sortB;
    }

    paA = el -> parentA;
    paB = el -> parentB;

    e_n = el -> nextA;
    e_p = el -> prevA;
    if (e_p) *ANEXT (e_p, paA) = e_n;
    else {
        if (el -> type == 'C')
            paA -> cap[sortA] = e_n;
        else
            paA -> con[sortA] = e_n;
    }
    if (e_n) *APREV (e_n, paA) = e_p;

    e_n = el -> nextB;
    e_p = el -> prevB;
    if (e_p) *ANEXT (e_p, paB) = e_n;
    else {
        if (el -> type == 'C')
            paB -> cap[sortB] = e_n;
        else
            paB -> con[sortB] = e_n;
    }
    if (e_n) *APREV (e_n, paB) = e_p;

    if (el -> type == 'C') {
        currIntCap--;
        nqChange (paA, 0, -1);
        nqChange (paB, 0, -1);
        hashval = ELEMHASH (paA, paB, capHT_size);
        e_n = capHashTab[hashval];
    }
    else {
        currIntRes--;
        nqChange (paA, -1, 0);
        nqChange (paB, -1, 0);
        hashval = ELEMHASH (paA, paB, resHT_size);
        e_n = resHashTab[hashval];
    }

    e_p = NULL;
```

```
    while (e_n) {
        if (e_n == el) break;
        e_n = (e_p = e_n) -> nextHash;
    }
    ASSERT (e_n == el);

    if (e_p) e_p -> nextHash = el -> nextHash;
    else {
        if (el -> type == 'C')
            capHashTab[hashval] = el -> nextHash;
        else
            resHashTab[hashval] = el -> nextHash;
    }

    // elemDispose (el);
    if (el -> moments) DISPOSE (el -> moments);
    CLEAR (el, sizeof (element_t));
    el -> nextA = elemList;
    elemList = el;
}

// see: lump/node.c

void nqChange (n, dr, dc) node_t *n; int dr, dc;
{
    if (n -> delayed) nqDelete (n); // the delayed position is based on the node degree
    n -> res_cnt += dr;
    n -> cap_cnt += dc;
    n -> degree = nqDegree (n); // calculation method is based on elim_order
    if (n -> delayed) nqInsert (n);
}

// see: lump/init.c

void initLump (cellCirKey, cellLayKey, scale)
{
    inCap = outCap = currIntCap = maxIntCap = maxCapIfKept = 0;
    inRes = outRes = currIntRes = maxIntRes = maxResIfKept = 0;
    ...
}

void endLump ()
{
    ...
    if (currIntCap > 0 || currIntRes > 0 || ...)
        fprintf (stderr, "\nWARNING: Circuit items left behind in core!\n\n");

    if (lastPass && optInfo) {
        printf ("\nextraction statistics for layout %s:\n", layoutName);
        printf ("                in    out  maxIntern IfKept remaining (flushed)\n");
        printf ("capacitances  : %6d %6d %6d %6d %10d\n",
                inCap, outCap, maxIntCap, maxCapIfKept, currIntCap);
        printf ("resistances   : %6d %6d %6d %6d %10d\n",
                inRes, outRes, maxIntRes, maxResIfKept, currIntRes);
    }
    else if (lastPass && optVerbose) {
        printf ("\nextraction statistics for layout %s:\n", layoutName);
        printf ("        capacitances        : %d\n", outCap);
        printf ("        resistances         : %d\n", outRes);
        ...
    }
}
```