Magic Maintainer's Manual #2: The Technology File

Walter Scott

Special Studies Program
Lawrence Livermore National Laboratory
PO Box 808, L-270
Livermore, CA 94550
wss@mordor.s1.gov

John Ousterhout

Computer Science Division
Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720

(Updated by various folks.)

This tutorial corresponds to Magic version 6; technology suffix ''.tech26''

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #6: Design-Rule Checking

You should also read at least the first, and probably all four, of the papers on Magic that appeared in the *ACM IEEE 21st Design Automation Conference,* and the paper ''Magic's Circuit Extractor'', which appeared in the *ACM IEEE 22nd Design Automation Conference*. The overview paper from the DAC was also reprinted in *IEEE Design and Test* magazine in the February 1985 issue. The circuit extractor paper also appeared in the February 1986 issue of *IEEE Design and Test* magazine.

Commands covered in this tutorial:

:*watch

Macros covered in this tutorial:

none

Changes since Magic version 4:

Support for stacked contacts and more than two metal layers, causing slight changes in the semantics of **contact** definitions

Support for new Magic maze router, *Mzrouter* (see Tutorial #10)

Support for new gate-array router, *Garouter* (see Tutorial #12)

Support for extractor extensions (see Tutorial #8)

Chunk sizes can now be specified explicitly for DRC and CIF output

Unlike CIF, calma scalefactors needn't be multiples of 2 any more

1. Introduction

Magic is a technology independent layout editor. All technology-specific information—mask layers, design rules, etc.—comes from a *technology file*. There is a different technology file for each technology supported by Magic. You can run Magic with a different technology by specifying the **-T***techfile* flag on the command line you use to start Magic, where *techfile* is the name of a file of the form *techname***.tech***n* in either the current directory, or the library directory ~cad/lib/magic/sys. (The *n* is a numeric suffix to identify the version of the technology file, which is currently **26**).

This tutorial describes the contents of a technology file, and gives hints for building a new one. It assumes that your current working directory is the Magic source directory, either ~**cad/src/magic** or **${CAD_HOME}/src/magic**.

A technology file is organized into sections, each of which begins with a line containing a single keyword and ends with a line containing the single word **end**. If you examine one of the Magic technology files in the directory ~**cad/src/magic/tech**, e.g, **scmos.tech**, you can see that it contains the following sections: **tech**, **planes**, **types**, **styles**, **contact**, **compose**, **connect**, **cifoutput**, **cifinput**, **mzrouter**, **drc**, **extract**, **wiring**, **router, plowing**, and **plot**. These sections must appear in this order in all technology files. Every technology file must have all of the sections, although the sections need not have any lines between the section header and the **end** line.

A technology file can contain comments, which are blocks of text beginning with the characters ‘‘**/***’’ and ending with the characters ‘‘***/**’’. Comments are ignored when processing a technology file. In **scmos.tech** you can see several lines just before the **connect** section (near the beginning of the technology file) that are of the form ‘‘**#define** ...’’. These lines are definitions of macros that may be used in subsequent lines in the technology file.

The form of comments and macro definitions should look familiar to ‘‘C’’ programmers, for good reason: the ‘‘C’’ macro preprocessor is used to expand macros and eliminate comments. Technology files cannot be read directly by Magic in their ‘‘raw’’ form; the ‘‘C’’ preprocessor is run to produce a Magic-readable version of the technology file. The last section in this tutorial describes how to install technology files.

Each section in a technology file consists of a series of lines. Each line consists of a series of words, separated by spaces or tabs. If a line ends with the character ‘‘\’’, the ‘‘\’’ is ignored and the following newline is treated as an ordinary blank. For example,

```
width    allDiff    2 \
``Diffusion width must be at least 2''
```

is treated as though it had all appeared on a single line with no intervening ``\''. The rest of this part of the tutorial will describe each of the technology file sections in turn.

2. Tech section

Magic stores the technology of a cell in the cell's file on disk. When reading a cell back in to Magic from disk, the cell's technology must match the name of the current technology, which appears as a single word in the **tech** section of the technology file. See Table 1 for an example.

tech
scmos
end

Table 1. **Tech** section

It may seem that storing the technology name as part of the technology file is redundant, since the name of the file is probably the same as the name of the technology. (e.g., ``scmos.tech26'' for technology **scmos**, or ``nmos.tech26'' for technology **nmos**). This feature is leftover from olden days when slight variants of a technology would be created by having a new technology file with a different file name but the same ``official'' name given in the **tech** section. This has the advantage that cells designed with one variant could be edited with any of the other files implementing the same technology without having to modify the technology names in the **.mag** files. The disadvantage of this approach, however, is that it defeats Magic technology-defaulting mechanism: if no explicit technology is specified when Magic starts up, it reads the technology from the **.mag** file being edited and looks for a technology file by this name. If there are several variants of the same technology, Magic will pick the one with the desired technology file name. Anyhow, we recommend that the internal names of technologies should always match the file names.

3. Planes, types, and contact sections

The **planes**, **types**, and **contact** sections are used to define the layers used in the technology. Magic uses a new data structure, called *corner-stitching*, to represent layouts. Corner-stitching represents mask information as a collection of non-overlapping rectangular *tiles*. Each tile has a type that corresponds to a single Magic layer. An individual corner-stitched data structure is referred to as a *plane*.

Magic allows you to see the corner-stitched planes it uses to store a layout. We'll use this facility to see how several corner-stitched planes are used to store the layers of a layout. Enter Magic to edit the cell **m2a**. Type the command **:*watch active demo**.

You are now looking at the **active** plane. Each of the boxes outlined in black is a tile. (The arrows are *stitches*, but are unimportant to this discussion.) You can see that some tiles contain layers (polysilicon, ndiffusion, ndcontact, polycontact, and ntransistor), while others contain empty space. Corner-stitching is unusual in that it represents empty space explicitly. Each tile contains exactly one type of material, or space.

You have probably noticed that metal1 does not seem to have a tile associated with it, but instead appears right in the middle of a space tile. This is because metal1 is stored on a different plane, the **metal1** plane. Type the command **:*watch metal1 demo**. Now you can see that there are metal1 tiles, but the polysilicon, diffusion, and transistor tiles have disappeared. The two contacts, polycontact and ndcontact, still appear to be tiles.

The reason Magic uses several planes to store mask information is that corner-stitching can only represent non-overlapping rectangles. If a layout were to consist of only a single layer, such as polysilicon, then only two types of tiles would be necessary: polysilicon and space. As more layers are added, overlaps can be represented by creating a special tile type for each kind of overlap area. For example, when polysilicon overlaps ndiffusion, the overlap area is marked with the tile type ntransistor.

Although some overlaps correspond to actual electrical constructs (e.g., transistors), other overlaps have little electrical significance. For example, metal1 can overlap polysilicon without changing the connectivity of the circuit or creating any new devices. The only consequence of the overlap is possibly a change in parasitic capacitance. To create new tile types for all possible overlapping combinations of metal1 with polysilicon, diffusion, transistors, etc. would be wasteful, since these new overlapping combinations would have no electrical significance.

Instead, Magic partitions the layers into separate planes. Layers whose overlaps have electrical significance must be stored in a single plane. For example, polysilicon, diffusion, and their overlaps (transistors) are all stored in the **active** plane. Metal1 does not interact with any of these tile types, so it is stored in its own plane, the **metal1** plane. Similarly, in the scmos technology, metal2 doesn't interact with either metal1 or the active layers, so is stored in yet another plane, **metal2**.

Contacts between layers in one plane and layers in another are a special case and are represented on *both* planes. This explains why the pcontact and ndcontact tiles appeared on both the **active** plane and on the **metal1** plane. Later in this section, when the **contacts** section of the technology file is introduced, we'll see how to define contacts and the layers they connect.

The **planes** section of the technology file specifies how many planes will be used to store tiles in a given technology, and gives each plane a name. Each line in this section defines a plane by giving a comma-separated list of the names by which it is known. Any name may be used in referring to the plane in later sections, or in commands like the **:*watch** command you used earlier. Table 2 gives the **planes** section from the scmos technology file.

Magic uses six other planes internally. The **subcell** plane is used for storing cell instances rather than storing mask layers. The **designRuleCheck** and **designRuleError** planes are used by the design rule checker to store areas to be reverified, and areas containing design rule violations, respectively. Finally, the **mhint**, **fhint**, and **rhint** planes are used for by the interactive router (the **:iroute** command) for designer-specified

planes
well,w
active,diffusion,polysilicon,a
metal1,m1
metal2,m2
oxide,ox
end

Table 2. **Planes** section

graphic hints.

There is a limit on the maximum number of planes in a technology, including the internal planes. This limit is currently 15. To increase the limit, it is necessary to change **MAXPLANES** in the file **database/database.h** and then recompile all of Magic as described in ''Maintainer's Manual #1''. Each additional plane involves additional storage space in every cell and some additional processing time for searches, so we recommend that you keep the number of planes as small as you can do cleanly.

types	
active	polysilicon,red,poly,p
active	ndiffusion,green,ndiff
active	pdiffusion,brown,pdiff
metal1	metal1,m1,blue
metal2	metal2,m2,purple
well	pwell,pw
well	nwell,nw
active	polycontact,pcontact,pc
active	ndcontact,ndc
active	pdcontact,pdc
metal1	m2contact,m2c,via,v
active	ntransistor,nfet
active	ptransistor,pfet
active	psubstratepcontact,ppcontact,ppcont,psc,ppc,pwc,pwcontact
active	nsubstratencontact,nncontact,nncont,nsc,nnc,nwc,nwcontact
active	psubstratepdiff,psd,ppdiff,ppd,pohmic
active	nsubstratendiff,nsd,nndiff,nnd,nohmic
metal2	pad
oxide	glass
end	

Table 3. **Types** section

The **types** section identifies the technology-specific tile types used by Magic. Table 3 gives this section for the scmos technology file. Each line in this section is of the following form:

plane names

Each type defined in this section is allowed to appear on exactly one of the planes defined in the **planes** section, namely that given by the *plane* field above. For contacts types such as **pcontact**, the plane listed is considered to be the contact's *home* plane. Other tile types will be used to represent the contact on the other planes it connects; this is described later in this section.

The *names* field is a comma-separated list of names. The first name in the list is the ‘‘long’’ name for the type; it appears in the **.mag** file and whenever error messages involving that type are printed. Any unique abbreviation of any of a type's names is sufficient to refer to that type, both from within the technology file and in any commands such as **:paint** or **:erase**.

Tile type	Plane
space	*all*
error_p, EP	**designRuleError**
error_s, ES	**designRuleError**
error_ps, EPS	**designRuleError**
checkpaint, CP	**designRuleCheck**
checksubcell, CS	**designRuleCheck**
magnet, mag	**mhint**
fence, f	**fhint**
rotate, r	**rhint**

Table 4. Built-in Magic types.

Magic has certain built-in types as shown in Table 4. Empty space (**space**) is special in that it can appear on any plane. The types **error_p**, **error_s**, and **error_ps** record design rule violations. The types **checkpaint** and **checksubcell** record areas still to be design-rule checked. **Magnet**, **fence**, and **rotate** are the types used by designers to indicate hints for the irouter.

There is a limit on the maximum number of types in a technology, including all the built-in types. Currently, the limit is 80 tile types. To increase the limit, you'll have to change **TT_MAXTYPES** in the file **database/database.h** and then recompile all of Magic as described in ‘‘Maintainer's Manual #1’’. A number of macros in **database.h** also depend on the value of **TT_MAXTYPES**/32. They are currently set up assuming that **TT_MAXTYPES** is between 65 and 96; if **TT_MAXTYPES** is changed to lie outside this region they should be changed. See the comments in **database.h** for more information. Because there are a number of tables whose size is determined by the square of **TT_MAXTYPES**, it is very expensive to increase **TT_MAXTYPES** much beyond the current limit.

contact			
pcontact	poly	metal1	
ndcontact	ndiff	metal1	
pdcontact	pdiff	metal1	
ppcontact	ppdiff	metal1	
nncontact	nndiff	metal1	
m2contact	metal2	metal1	
pad	metal1	metal2	glass
end			

Table 5. **Contact** section

As mentioned above, contacts in Magic are represented on each plane containing material connected by the contact. Also mentioned above, though, each tile type defined in the **types** section appears on exactly one plane. This seeming conflict is resolved by having Magic automatically generate new tile types for each of the planes on which a contact appears. The **contact** section lets Magic know which types are contacts, and the adjacent planes and component types to which they are connected.

Each line in the **contact** section begins with a tile type, *base*, which is thereby defined to be a contact. This type is also referred to as a contact's *base type*. The remainder of each line is a list of two or three non-contact tile types that are connected by the contact. These tile types are referred to as the *component* types of the contact, and are the layers that would be present if there were no electrical connection (*i.e.*, no via hole). In Table 5, for example, the type **pcontact** is the base type of a contact connecting the component layers **polysilicon** on the active plane with **metal1** on the metal1 plane.

The home plane of one of the component types should be the same as that of the base type (**active** in the case of **pcontact**), and the other type(s) must be on planes adjacent to that of the base. (If two planes appear on subsequent lines in the **planes** section, then they are considered to be adjacent. Hence the **active** plane is adjacent to the **well** and **metal1** planes, but not to the **metal2** plane.)

The above scheme allows you to define contacts that connect at most three layers together. For example, if the scmos technology supported stacked contacts (it doesn't), you could define a contact (the type **pm12c**) that connected polysilicon, metal1, and metal2 as:

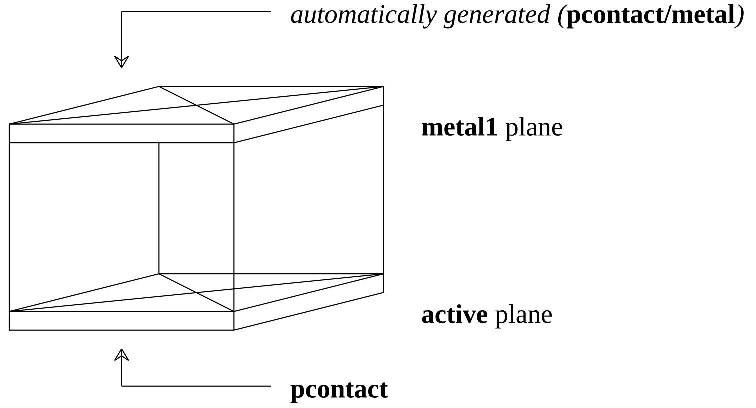


Figure 1. A different tile type is used to represent a contact on each plane that it connects. Here, a contact between poly on the **active** plane and metal1 on the **metal1** plane is stored as two tile types. One, **pcontact**, is specified in the technology file as residing on the **active** plane; the other is automatically-generated for the **metal1** plane.

contact pm12c polysilicon metal1 metal2

but it wouldn't be possible to define directly a contact that connected four layers, such as pwell, ndiffusion, metal1, and metal2. To define this latter type of contact, you'd have to define two different contact tile types, **wndc** and **nm12c**, and use two **contact** lines, each of which connects only three layers.

contact wndc pwell ndiffusion metal1
contact nm12c ndiffusion metal1 metal2

Magic will automatically generate the appropriate paint and erase rules to allow painting, for example, **nm12c** over **wndc** to give the four-layer stacked contact. In order for Magic to do this, however, you must be certain never to define two types of contacts with identical component types. This is the reason why, in the scmos technology file, the **pad** contact is shown as connecting three layers (**metal1**, **metal2**, and **glass**) instead of just **metal1** and **metal2** as it used to with version 4 of Magic.

Each contact has an *image* tiletype on all the planes it connects. Normally, this means that Magic has to generate a new tiletype for all the planes of a contact other than its base plane. In the first example, this means that a new tile type will be generated to represent **pcontact** on the metal1 plane. The type used to represent the contact on the active plane is **pcontact** itself. Figure 1 depicts the situation graphically. In later sections of the technology file, it is sometimes useful to refer separately to the various images of contact. A special notation using a ''/'' is used for this. If a tile type *aaa/bbb* is specified in the technology file, this refers to the image of contact *aaa* on plane *bbb*. For example, **pcontact/metal1** refers to the image of the pcontact that lies on the metal1 plane, and **pcontact/active** refers to the image on the active plane, which is the same as **pcontact**.

Sometimes, however, there is already an existing contact that has the required connectivity to be an image of a new contact. For example, the **pad** type connects **metal1**, **metal2**, and **glass**. Its home plane is **metal2**, which implies that we need images on the **metal1** and **oxide** planes that have the required connectivity. There's no existing contact between **glass** and **metal2**, so the image on the **oxide** plane must be generated automatically by Magic. However, there is already a contact that connects the component types **metal1** and **metal2**, namely **m2c**, so this is used as the image of **pad** on the **metal1** plane.

4. Specifying Type-lists

In several places in the technology file you'll need to specify groups of tile types. For example, in the **connect** section you'll specify groups of tiles that are mutually connected. These are called *type-lists* and there are several ways to specify them. The simplest form for a type-list is a comma-separated list of tile types, for example

poly,ndiff,pcontact,ndc

The null list (no tiles at all) is indicated by zero, i.e.,

0

There must not be any spaces in the type-list. Type-lists may also use tildes (‘‘~’’) to select all tiles but a specified set, and parentheses for grouping. For example,

~(pcontact,ndc)

selects all tile types but pcontact and ndc. When a contact name appears in a type-list, it selects *all* images of the contact unless a ‘‘/’’ is used to indicate a particular one. The example above will not select any of the images of pcontact or ndc. Slashes can also be used in conjunction with parentheses and tildes. For example,

~(pcontact,ndc)/active,metal1

selects all of the tile types on the active plane except for pcontact and ndc, and also selects metal1. Tildes have higher operator precedence than slashes, and commas have lowest precedence of all.

Note: in the CIF sections of the technology file, only simple comma-separated names are permitted; tildes and parentheses are not understood. However, everywhere else in the technology file the full generality can be used. Sorry for this inconsistency...

5. Styles section

Magic can be run on several different types of graphical displays. Although it would have been possible to incorporate display-specific information into the technology file, a different technology file would have been required for each display type. Instead, the technology file gives one or more display-independent *styles* for each type that is to be displayed, and uses a per-display-type styles file to map to colors and stipplings specific to the display being used. The styles file is described in Magic Maintainer's Manual #3: ‘‘Styles and Colors’’, so we will not describe it further here.

styles	
styles	
styletype mos	
poly	1
ndiff	2
pdiff	4
nfet	6
nfet	7
pfet	8
pfet	9
metal1	20
metal2	21
pcontact	1
pcontact	20
pcontact	32
ndcontact	2
ndcontact	20
ndcontact	32
pdcontact	4
pdcontact	20
pdcontact	32
m2contact	20
m2contact	21
m2contact	33
end	

Table 6. Part of the **styles** section

Table 6 shows part of the **styles** section from the scmos technology file. The first line specifies the type of style file for use with this technology, which in this example is **mos**. Each subsequent line consists of a tile type and a style number (an integer between 1 and 63). The style number is nothing more than a reference between the technology file and the styles file. Notice that a given tile type can have several styles (e.g., pcontact uses styles #1, #20, and #32), and that a given style may be used to display several different tiles (e.g., style #2 is used in ndiff and ndcontact). If a tile type should not be displayed, it has no entry in the **styles** section.

6. Compose section

The semantics of Magic's paint operation are defined by a collection of rules of the form, ‘‘given material *HAVE* on plane *PLANE*, if we paint *PAINT*, then we get *Z*’’, plus a similar set of rules for the erase operation. The default paint and erase rules are simple. Assume that we are given material *HAVE* on plane *PLANE*, and are painting or erasing

compose			
compose	nfet	poly	ndiff
compose	pfet	poly	pdiff
paint	pwell	nwell	nwell
paint	nwell	pwell	pwell
paint	pdc/active	pwell	ndc/active
paint	pdc/m1	pwell	ndc/m1
paint	pfet	pwell	nfet
paint	pdiff	pwell	ndiff
paint	nsd	pwell	psd
paint	nsc/active	pwell	psc/active
paint	nsc/m1	pwell	psc/m1
paint	ndc/active	nwell	pdc/active
paint	ndc/m1	nwell	pdc/m1
paint	nfet	nwell	pfet
paint	ndiff	nwell	pdiff
paint	psd	nwell	nsd
paint	psc/active	nwell	nsc/active
paint	psc/m1	nwell	nsc/m1
end			

Table 7. **Compose** section

material *PAINT*.

(1) *You get what you paint.* If the home plane of *PAINT* is *PLANE*, or *PAINT* is space, you get *PAINT*; otherwise, nothing changes and you get *HAVE*.

(2) *You can erase all or nothing.* Erasing space or *PAINT* from *PAINT* will give space; erasing anything else has no effect.

These rules apply for contacts as well. Painting the base type of a contact paints the base type on its home plane, and each image type on its home plane. Erasing the base type of a contact erases both the base type and the image types.

It is sometimes desirable for certain tile types to behave as though they were ‘‘composed’’ of other, more fundamental ones. For example, painting poly over ndiffusion in scmos produces ntransistor, instead of ndiffusion. Also, painting either poly or ndiffusion over ntransistor leaves ntransistor, erasing poly from ntransistor leaves ndiffusion, and erasing ndiffusion leaves poly. The semantics for ntransistor are a result of the following rule in the **compose** section of the scmos technology file:

compose ntransistor poly ndiff

Sometimes, not all of the ‘‘component’’ layers of a type are layers known to magic. As an example, in the **nmos** technology, there are two types of transistors: **enhancement-fet** and **depletion-fet**. Although both contain polysilicon and diffusion, depletion-fet can be thought of as also containing implant, which is not a tile type. So while we can't construct depletion-fet by painting poly and then diffusion, we'd still like it to behave as though it contained both materials. Painting poly or diffusion over a depletion-fet should not change it, and erasing either poly or diffusion should give the other. These semantics are the result of the following rule:

> **decompose** dfet poly diff

The general syntax of both types of composition rules, **compose** and **decompose**, is:

> **compose** *type a1 b1 a2 b2 ...*
> **decompose** *type a1 b1 a2 b2 ...*

The idea is that each of the pairs *a1 b1, a2 b2*, etc comprise *type*. In the case of a **compose** rule, painting any *a* atop its corresponding *b* will give *type*, as well as vice-versa. In both **compose** and **decompose** rules, erasing *a* from *type* gives *b*, erasing *b* from *type* gives *a*, and painting either *a* or *b* over *type* leaves *type* unchanged.

Contacts are implicitly composed of their component types, so the result obtained when painting a type *PAINT* over a contact type *CONTACT* will by default depend only on the component types of *CONTACT*. If painting *PAINT* doesn't affect the component types of the contact, then it is considered not to affect the contact itself either. If painting *PAINT* does affect any of the component types, then the result is as though the contact had been replaced by its component types in the layout before type *PAINT* was painted. Similar rules hold for erasing.

A pcontact has component types poly and metal1. Since painting poly doesn't affect either poly or metal1, it doesn't affect a pcontact either. Painting ndiffusion does affect poly—it turns it into an ntransistor—. Hence, painting ndiffusion over a pcontact breaks up the contact, leaving ntransistor on the active plane and metal1 on the metal1 plane.

The **compose** and **decompose** rules are normally sufficient to specify the desired semantics of painting or erasing. In unusual cases, however, it may be necessary to provide Magic with explicit **paint** or **erase** rules. For example, to specify that painting pwell over pdiffusion switches its type to ndiffusion, the technology file contains the rule:

> **paint** pdiffusion pwell ndiffusion

This rule could not have been written as a **decompose** rule; erasing ndiffusion from pwell does not yield pdiffusion, nor does erasing pdiffusion from ndiffusion yield pwell. The general syntax for these explicit rules is:

> **paint** *have t result* [*p*]
> **erase** *have t result* [*p*]

Here, *have* is the type already present, on plane *p* if it is specified; otherwise, on the home plane of *have*. Type *t* is being painted or erased, and the result is type *result*. Table 7 gives the **compose** section for scmos.

It's easiest to think of the paint and erase rules as being built up in four passes. The first pass generates the default rules for all non-contact types, and the second pass replaces these as specified by the **compose**, **decompose**, etc. rules, also for non-contact types. At this point, the behavior of the component types of contacts has been completely determined, so the third pass can generate the default rules for all contact types, and the fourth pass can modify these as per any **compose**, etc. rules for contacts.

```
connect
#define allMetal2 m2,m2c/m2,pad/m2
#define allMetal1 m1,m2c/m1,pc/m1,ndc/m1,pdc/m1,ppcont/m1,nncont/m1,pad/m1
#define allPoly poly,pc/a,nfet,pfet
allMetal2        allMetal2
allMetal1        allMetal1
allPoly          allPoly
ndiff            ndc
pdiff            pdc
nwell,nnc,nsd    nwell,nnc,nsd
pwell,ppc,psd    pwell,ppc,psd
nnc              pdc
ppc              ndc
end
```

Table 8. **Connect** section

7. Connect section

For circuit extraction, routing, and some of the net-list operations, Magic needs to know what types are electrically connected. Magic's model of electrical connectivity used is based on signal propagation. Two types should be marked as connected if a signal will *always* pass between the two types, in either direction. For the most part, this will mean that all non-space types within a plane should be marked as connected. The exceptions to this rule are devices (transistors). A transistor should be considered electrically connected to adjacent polysilicon, but not to adjacent diffusion. This models the fact that polysilicon connects to the gate of the transistor, but that the transistor acts as a switch between the diffusion areas on either side of the channel of the transistor.

The lines in the **connect** section of a technology file, as shown in Table 8, each contain a pair of type-lists in the format described in Section 4. Each type in the first list connects to each type in the second list. This does not imply that the types in the first list are themselves connected to each other, or that the types in the second list are connected to each other.

Because connectivity is a symmetric relationship, only one of the two possible orders of two tile types need be specified. Tiles of the same type are always considered to be connected. Contacts are treated specially; they should be specified as connecting to material in all planes spanned by the contact. For example, pcontact is shown as connecting to several types in the active plane, as well as several types in the metal1 plane. The connectivity of a contact should usually be that of its component types, so pcontact should connect to everything connected to poly, and to everything connected to metal1.

```
cifoutput
style lambda=1.0(gen)
    scalefactor 100
    layer CWN nwell
        bloat-or pdiff,pdc,pfet * 600
        bloat-or nsc,nnd * 300
        grow 300
        shrink 300
        calma 42 1
    layer CWP pwell
        bloat-or ndiff,ndc,nfet * 600
        bloat-or psc,ppd * 300
        grow 300
        shrink 300
        calma 41 1
    layer CMS allMetal2
        labels m2
        calma 51 1
    layer CAA allDiff
        labels ndiff,pdiff
        calma 43 1
    layer CCA ndc,pdc
        squares 200
        calma 48 1
    layer CCA nncont,ppcont
        squares 200
        calma 48 1
    layer CCP pc
        squares 200
        calma 47 1
end
```

Table 9. Part of the **cifoutput** section for style lambda=1.0(gen) only

8. Cifoutput section

The layers stored by Magic do not always correspond to physical mask layers. For example, there is no physical layer corresponding to ntransistor; instead, the actual circuit must be built up by overlapping poly and diffusion over pwell. When writing CIF (Caltech Intermediate Form) or Calma GDS-II files, Magic generates the actual geometries that will appear on the masks used to fabricate the circuit. The **cifoutput** section of the technology file describes how to generate mask layers from Magic's abstract layers.

8.1. CIF styles

The technology file can contain several different specifications of how to generate CIF. Each of these is called a CIF *style*. Different styles may be used for fabrication at different feature sizes, or for totally different purposes. For example, some of the Magic technology files contain a style ''plot'' that generates CIF pseudo-layers that have exactly the same shapes as the Magic layers. This style is used for generating plots that look just like what appears on the color display; it makes no sense for fabrication. Lines of the form

style *name*

are used to end the description of the previous style and start the description of a new style. The Magic command **:cif ostyle** *name* is typed by users to change the current style used for output. The first style in the technology file is used by default for CIF output if the designer doesn't issue a **:cif style** command. If the first line of the **cifoutput** section isn't a **style** line, then Magic uses an initial style name of **default**.

8.2. Scaling

Each style must contain a line of the form

scalefactor *scale* **[***reducer*]

that tells how to scale Magic coordinates into CIF coordinates. The argument *scale* indicates how many hundredths of a micron correspond to one Magic unit. Because of certain numerical problems with the CIF representation, *scale* must always be an even number (except as described below).

The second parameter, *reducer*, is optional. If it is specified, it may be either the keyword **calmaonly**, or an integer. If *reducer* is **calmaonly**, then this output style can only be used to generate Calma (GDS-II) output, not CIF, but the restriction that *scale* must always be an even number is relaxed; *scale* can be any positive integer.

If *reducer* is an integer, it is used to increase the readability and decrease the size of CIF files. Each CIF coordinate is divided by *reducer* before being written to the CIF file, then a uniform upward scalefactor of *reducer* is specified once for the whole file. This has no effect on the CIF except to make the individual CIF numbers smaller and thereby reduce the sizes of CIF files. *Reducer* must be a positive integer, and must evenly divide into every other dimension specified in any statement for this style. *Reducer* must also divide one-half of *scale*. If this sounds confusing, the easiest thing is to leave *reducer* unspecified, in which case the value 1 is used.

In addition to specifying a scale factor, each style can specify the size in which chunks will be processed when generating CIF hierarchically. This is particularly important when the average design size is much larger than the maximum bloat or shrink (e.g, more than 3 orders of magnitude difference). The step size is specified by a line of the following form:

stepsize *stepsize*

where *stepsize* is in Magic units. For example, if you plan to generate CIF for designs that will typically be 100,000 Magic units on a side, it might make sense for *stepsize* to be 10000 or more.

8.3. Layer descriptions

The main body of information for each CIF style is a set of layer descriptions. Each layer description consists of one or more lines describing how to generate the CIF for a single layer. The first line of each description is one of

layer *name* [*layers*]
or
templayer *name* [*layers*]

These statements are identical, except that templayers are not output in the CIF file. They are used only to build up intermediate results used in generating the ''real'' layers. In each case, *name* is the CIF name to be used for the layer. If *layers* is specified, it consists of a comma-separated list of Magic layers and previously-defined CIF layers in this style; these layers form the initial contents of the new CIF layer (note: the layer lists in this section are less general than what was described in Section 4; tildes and parentheses are not allowed). If *layers* is not specified, then the new CIF layer is initially empty. The following statements are used to modify the contents of a CIF layer before it is output.

After the **layer** or **templayer** statement come several statements specifying geometrical operations to apply in building the CIF layer. Each statement takes the current contents of the layer, applies some operation to it, and produces the new contents of the layer. The last geometrical operation for the layer determines what is actually output in the CIF file. The geometrical operations are:

or *layers*
and *layers*
and-not *layers*
grow *amount*
shrink *amount*
bloat-or *layers layers2 amount layers2 amount ...*
bloat-max *layers layers2 amount layers2 amount ...*
bloat-min *layers layers2 amount layers2 amount ...*
squares *size*
squares *border size separation*

The operation **or** takes all the *layers* (which may be either Magic layers or previously-defined CIF layers), and or's them with the material already in the CIF layer. The operation **and** is similar to **or**, except that it and's the layers with the material in the CIF layer (in other words, any CIF material that doesn't lie under material in *layers* is removed from the CIF layer). **And-not** finds all areas covered by *layers* and erases current CIF material from those areas. **Grow** and **shrink** will uniformly grow or shrink the current CIF layer by *amount* units, where *amount* is specified in CIF units, not Magic units.

The three **bloat** operations provide selective forms of growing. In these statements, all the layers must be Magic layers. Each operation examines all the tiles in *layers*, and grows the tiles by a different distance on each side, depending on the rest of the line. Each pair *layers2 amount* specifies some tile types and a distance (in CIF units). Where a

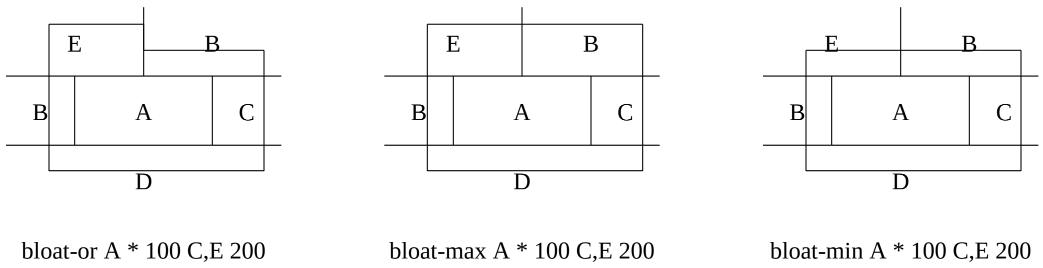


Figure 2. The three different forms of **bloat** behave slightly differently when two different bloat distances apply along the same side of a tile. In each of the above examples, the CIF that would be generated is shown in bold outline. If **bloat-or** is specified, a jagged edge may be generated, as on the left. If **bloat-max** is used, the largest bloat distance for each side is applied uniformly to the side, as in the center. If **bloat-min** is used, the smallest bloat distance for each side is applied uniformly to the side, as on the right.

tile of type *layers* abuts a tile of type *layers2*, the first tile is grown on that side by *amount*. The result is or'ed with the current contents of the CIF plane. The layer ''*'' may be used as *layers2* to indicate all tile types. Where tiles only have a single type of neighbor on each side, all three forms of **bloat** are identical. Where the neighbors are different, the three forms are slightly different, as illustrated in Figure 2. Note: all the layers specified in any given **bloat** operation must lie on a single Magic plane. For **bloat-or** all distances must be positive. In **bloat-max** and **bloat-min** the distances may be negative to provide a selective form of shrinking.

In retrospect, it's not clear that **bloat-max** and **bloat-min** are very useful operations. The problem is that they operate on tiles, not regions. This can cause unexpected behavior on concave regions. For example, if the region being bloated is in the shape of a ''T'', a single bloat factor will be applied to the underside of the horizontal bar. If you use **bloat-max** or **bloat-min**, you should probably specify design-rules that require the shapes being bloated to be convex.

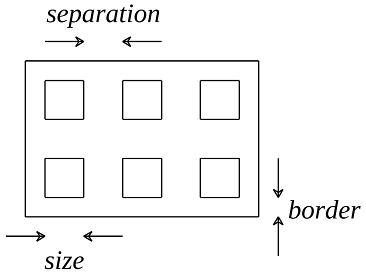


Figure 3. The **squares** operator chops each tile up into squares, as determined by the *border*, *size*, and *separation* parameters. In the example, the bold lines show the CIF that would be generated by a **squares** operation. The squares of material are always centered so that the borders on opposite sides are the same.

The last geometric operation is called **squares**. It examines each tile on the CIF plane, and replaces that tile with one or more squares of material. Each square is *size* CIF units across, and squares are separated by *separation* units. A border of at least *border* units is left around the edge of the original tile, if possible. This operation is used to generate contact vias, as in Figure 3. If only one argument is given in the **squares** statement, then *separation* defaults to *size* and *border* defaults to *size*/2. If a tile doesn't hold an integral number of squares, extra space is left around the edges of the tile and the squares are centered in the tile. If the tile is so small that not even a single square can fit and still leave enough border, then the border is reduced. If a square won't fit in the tile, even with no border, then no material is generated. The **squares** operation must be used with some care, in conjunction with the design rules. For example, if there are several adjacent skinny tiles, there may not be enough room in any of the tiles for a square, so no material will be generated at all. Whenever you use the **squares** operator, you should use design rules to prohibit adjacent contact tiles, and you should always use the **no_overlap** rule to prevent unpleasant hierarchical interactions. The problems with hierarchy are discussed in Section 8.6 below, and design rules are discussed in Section 10.

8.4. Labels

There is an additional statement permitted in the **cifoutput** section as part of a layer description:

labels *Magiclayers*

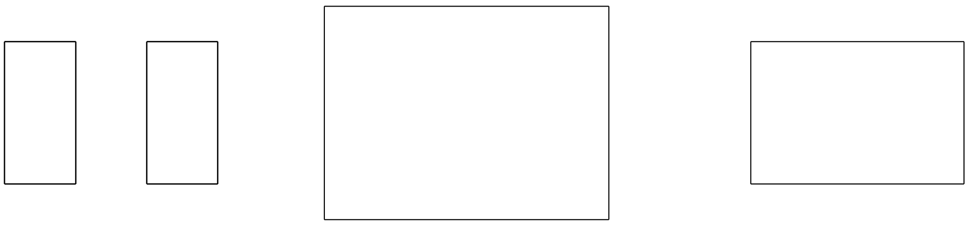
This statement tells Magic that labels attached to Magic layers *Magiclayers* are to be associated with the current CIF layer. Each Magic layer should only appear in one such statement for any given CIF style. If a Magic layer doesn't appear in any **labels** statement, then it is not attached to a specific layer when output in CIF.

8.5. Calma (GDS II Stream format) layers

Each layer description in the **cifoutput** section may also contain one of the following statements:

calma *calmaNumber calmaType*

This statement tells Magic which layer number and data type to use when the **calma** command outputs Calma GDS II Stream format for this CIF layer. Both *calmaNumber* and *calmaType* should be positive integers, between 0 and 63. Each CIF layer should have a different *calmaNumber*. If there is no **calma** line for a given CIF layer, then that layer will not be output by the **:calma** command.



(a) (b) (c)

Figure 4. If the operator **grow 100** is applied to the shapes in (a), the merged shape in (b) results. If the operator **shrink 100** is applied to (b), the result is (c). However, if the two original shapes in (a) belong to different cells, and if CIF is generated separately in each cell, the result will be the same as in (a). Magic handles this by outputting additional information in the parent of the subcells to fill in the gap between the shapes.

8.6. Hierarchy

Hierarchical designs make life especially difficult for the CIF generator. The CIF corresponding to a collection of subcells may not necessarily be the same as the sum of the CIF's of the individual cells. For example, if a layer is generated by growing and then shrinking, nearby features from different cells may merge together so that they don't shrink back to their original shapes (see Figure 4). If Magic generates CIF separately for each cell, the interactions between cells will not be reflected properly. The CIF generator attempts to avoid these problems. Although it generates CIF in a hierarchical representation that matches the Magic cell structure, it tries to ensure that the resulting CIF patterns are exactly the same as if the entire Magic design had been flattened into a single cell and then CIF were generated from the flattened design. It does this by looking in each cell for places where subcells are close enough to interact with each other or with paint in the parent. Where this happens, Magic flattens the interaction area and generates CIF for it; then Magic flattens each of the subcells separately and generates CIF for them. Finally, it compares the CIF from the subcells with the CIF from the flattened parent. Where there is a difference, Magic outputs extra CIF in the parent to compensate.

Magic's hierarchical approach only works if the overall CIF for the parent ends up covering at least as much area as the CIFs for the individual components, so all compensation can be done by adding extra CIF to the parent. In mathematical terms, this requires each geometric operation to obey the rule

$$\mathrm{Op}(A \cup B) \supseteq \mathrm{Op}(A) \cup \mathrm{Op}(B)$$

The operations **and**, **or**, **grow**, and **shrink** all obey this rule. Unfortunately, the **and-not**, **bloat**, and **squares** operations do not. For example, if there are two partially-overlapping tiles in different cells, the squares generated from one of the cells may fall in the separations between squares in the other cell, resulting in much larger areas of material than expected. There are two ways around this problem. One way is to use the design rules to prohibit problem situations from arising. This applies mainly to the **squares** operator. Tiles from which squares are made should never be allowed to overlap other such tiles in different cells unless the overlap is exact, so each cell will generate squares in the same place. You can use the **exact_overlap** design rule for this.

The second approach is to leave things up to the designer. When generating CIF, Magic issues warnings where there is less material in the children than the parent. The designer can locate these problems and eliminate the interactions that cause the trouble. Warning: Magic does not check the **squares** operations for hierarchical consistency, so you absolutely must use **exact_overlap** design rule checks! Right now, the **cifoutput** section of the technology is one of the trickiest things in the whole file, particularly since errors here may not show up until your chip comes back and doesn't work. Be extremely careful when writing this part!

9. Cifinput section

In addition to writing CIF, Magic can also read in CIF files using the **:cif read** *file* command. The **cifinput** section of the technology file describes how to convert from CIF mask layers to Magic tile types. In addition, it provides information to the Calma reader to allow it to read in Calma GDS II Stream format files. The **cifinput** section is very

```
cifinput
style lambda=1.0(gen)
    scalefactor 100
    layer m1 CMF
        labels CMF
    layer ndiff CSN
        and CAA
    layer nsd CWN
        and CSN
        and CAA
    layer nfet CPG
        and CAA
        and CSN
    layer ndc CCA
        grow 100
        and CAA
        and CWP
        and CSN
        and CMF
    layer nncont CCA
        grow 100
        and CAA
        and CSN
        and CWN
        and CMF
    calma CAA 1 *
    calma CCA 2 *
    calma CMF 4 *
    calma CPG 7 *
    calma CSN 8 *
    calma CWN 11 *
    calma CWP 12 *
end
```

Table 10. Part of the **cifinput** section. The order of the layers is important, since each Magic layer overrides the previous ones just as if they were painted by hand.

similar to the **cifoutput** section. It can contain several styles, with a line of the form

style *name*

used to end the description of the previous style (if any), and start a new CIF input style called *name*. If no initial style name is given, the name **default** is assigned. Each style must have a statement of the form

scalefactor *centimicrons*

to indicate how many hundredths of a micron correspond to one unit in Magic.

Like the **cifoutput** section, each style consists of a number of layer descriptions. A layer description contains one or more lines describing a series of geometric operations to be performed on CIF layers. The result of all these operations is painted on a particular Magic layer just as if the user had painted that information by hand. A layer description begins with a statement of the form

layer *magicLayer* [*layers*]

In the **layer** statement, *magicLayer* is the Magic layer that will be painted after performing the geometric operations, and *layers* is an optional list of CIF layers. If *layers* is specified, it is the initial value for the layer being built up. If *layers* isn't specified, the layer starts off empty. As in the **cifoutput** section, each line after the *layer* statement gives a geometric operation that is applied to the previous contents of the layer being built in order to generate new contents for the layer. The result of the last geometric operation is painted into the Magic database.

The geometric operations that are allowed in the **cifinput** section are a subset of those permitted in the **cifoutput** section:

or *layers*
and *layers*
and-not *layers*
grow *amount*
shrink *amount*

In these commands the *layers* must all be CIF layers, and the *amounts* are all CIF distances (centimicrons). As with the **cifoutput** section, layers can only be specified in simple comma-separated lists: tildes and slashes are not permitted.

When CIF files are read, all the mask information is read for a cell before performing any of the geometric processing. After the cell has been completely read in, the Magic layers are produced and painted in the order they appear in the technology file. In general, the order that the layers are processed is important since each layer will usually override the previous ones. For example, in the scmos tech file shown in Table 10 the commands for **ndiff** will result in the **ndiff** layer being generated not only where there is only ndiffusion but also where there are ntransistors and ndcontacts. The descriptions for **ntransistor** and **ndcontact** appear later in the section, so those layers will replace the **ndiff** material that was originally painted.

Labels are handled in the **cifinput** section just like in the **cifoutput** section. A line of the form

labels *layers*

means that the current Magic layer is to receive all CIF labels on *layers*. This is actually just an initial layer assignment for the labels. Once a CIF cell has been read in, Magic scans the label list and re-assigns labels if necessary. In the example of Table 10, if a label is attached to the CIF layer CPG then it will be assigned to the Magic layer **poly**. However, the polysilicon may actually be part of a poly-metal contact, which is Magic layer **pcontact**. After all the mask information has been processed, Magic checks the material underneath the layer, and adjusts the label's layer to match that material (**pcontact** in this case). This is the same as what would happen if a designer painted **poly** over an area, attached a label to the material, then painted **pcontact** over the area.

No hierarchical mask processing is done for CIF input. Each cell is read in and its layers are processed independently from all other cells; Magic assumes that there will not be any unpleasant interactions between cells as happens in CIF output (and so far, at least, this seems to be a valid assumption).

If Magic encounters a CIF layer name that doesn't appear in any of the lines for the current CIF input style, it issues a warning message and ignores the information associated with the layer. If you would like Magic to ignore certain layers without issuing any warning messages, insert a line of the form

ignore *cifLayers*

where *cifLayers* is a comma-separated list of one or more CIF layer names.

Calma layers are specified via **calma** lines, which should appear at the end of the **cifinput** section. They are of the form:

calma *cifLayer calmaLayers calmaTypes*

The *cifLayer* is one of the CIF types mentioned in the **cifinput** section. Both *calmaLayers* and *calmaTypes* are one or more comma-separated integers between 0 and 63. The interpretation of a **calma** line is that any Calma geometry whose layer is any of the layers in *calmaLayers*, and whose type is any of the types in *calmaTypes*, should be treated as the CIF layer *cifLayer*. Either or both of *calmaLayers* and *calmaTypes* may be the character ***** instead of a comma-separated list of integers; this character means *all* layers or types respectively. It is commonly used for *calmaTypes* to indicate that the Calma type of a piece of geometry should be ignored.

Just as for CIF, Magic also issues warnings if it encounters unknown Calma layers while reading Stream files. If there are layers that you'd like Magic to ignore without issuing warnings, assign them to a dummy CIF layer and ignore the CIF layer.

10. Mzrouter section

This section defines the layers and contacts available to the Magic maze router, *mzrouter,* and assigns default costs for each type. Default widths and spacings are derived from the **drc** section of the technology file (described below) but can be overridden in this section. Other mzrouter parameters, for example, search rate and width, can also be specified in this section. The syntax and function of the lines in the **mzrouter** section of the technology file are specified in the subsections below. Each set of specifications should be headed by a **style** line. **Routelayer** and **routecontact** specifications should precede references to them.

10.1. Styles

The mzrouter is currently used in two contexts, interactively via the **iroute** command, and as a subroutine to the garouter for stem generation. To permit distinct parameters for these two uses, the lines in the **mzrouter** section are grouped into *styles*. The

mzrouter					
style irouter					
layer	m2	32	64	256	1
layer	m1	64	32	256	1
layer	poly	128	128	512	1
contact	m2contact	metal1	metal2	1024	
contact	pcontact	metal1	poly	2056	
notactive	poly	pcontact			
style garouter					
layer	m2	32	64	256	1
layer	m1	64	32	256	1
contact	m2contact	metal1	metal2	1024	
end					

Table 11. Mzrouter section for the scmos technology.

lines pertaining to the irouter should be preceded by

style irouter

and those pertaining to the garouter should be preceded by the specification

style garouter

Other styles can be specified, but are currently not used. Table 11 shows the mzrouter section from the scmos technology.

10.2. Layers

Layer lines define the route-layers available to the maze router in that style. They have the following form:

layer *type hCost vCost jogCost hintCost*

Here *type* is the name of the tiletype of the layer and *hCost*, *vCost*, *jogCost* and *hintCost*, are non-negative integers specifying the cost per unit horizontal distance, cost per unit vertical distance, cost per jog, and cost per unit area of deviation from magnets, respectively. Route layers for any given style must lie in distinct planes.

10.3. Contacts

Contact lines specify the route-contacts available to the mzrouter in the current style. They have the following form:

contact *type routeLayer1 routeLayer2 cost*

Here *type* is the tiletype of the contact, *routeLayer1* and *routeLayer2* are the two layers connected by the contact, and *cost* is a nonnegative integer specifying the cost per contact.

10.4. Notactive

It maybe desirable to have a layer or contact available to the maze router, but default to off, i.e., not be used by the mzrouter until explicitly made active. Route-types (route-layers or route-contacts) can be made to default to off with the following specification:

notactive *route-type* ... [**route-typen**]

10.5. Search

The search **rate**, **width**, and **penalty** parameters can be set with a specification of the form:

search *rate width penalty*

Here *rate* and *width* are positive integers. And *penalty* is a positive rational (it may include a decimal point). See the irouter tutorial for a discussion of these parameters. (Note that **penalty** is a ''wizardly'' parameter, i.e., it is interactively set and examined via **iroute wizard** not **iroute search**). If no **search** line is given for a style, the overall mzrouter defaults are used.

10.6. Width

Appropriate widths for route-types are normally derived from the **drc** section of the technology file. These can be overridden with width specifications of the following form:

width *route-type width*

Here *width* is a positive integer.

10.7. Spacing

Minimum spacings between routing on a route-type and other types are derived from the design rules. These values can be overridden by explicit spacing specifications in the **mzrouter** section. Spacing specifications have the following form:

spacing *routetype type1 spacing1* ... [*typen spacingn*]

Spacing values must be nonnegative integers or **NIL**. The special type **SUBCELL** can be used to specify minimum spacing to unexpanded subcells.

11. Drc section

The design rules used by Magic's design rule checker come entirely from the technology file. We'll look first at two simple kinds of rules, **width** and and **spacing**. Most of the rules in the **drc** section are one or the other of these kinds of rules.

#define	allDiff	ndiff,pdiff,ndc/a,pdc/a,ppcont/a,nncont/a,pfet,nfet,psd,nsd
#define	extPoly	poly,pcontact
#define	extM1	metal1,pcontact/m1,ndc/m1,ppcont/m1,pdc/m1,nncont/m1
#define	extM2	metal2,m2contact/m2

Table 12a. Abbreviations for sets of tile types.

width	pwell	6	*''P-Well width must be at least 6 (MOSIS rule #1.1)''*
width	nwell	6	*''N-Well width must be at least 6 (MOSIS rule #1.1)''*
width	allDiff	2	*''Diffusion width must be at least 2 (MOSIS rule #2.1)''*
width	allPoly	2	*''Polysilicon width must be at least 2 (MOSIS rule #3.1)''*

Table 12b. Some width rules in the **drc** section.

11.1. Width rules

The minimum width of a collection of types, taken together, is expressed by a **width** rule. Such a rule has the form:

width *type-list width error*

where *type-list* is a set of tile types (see Section 4 for syntax), *width* is an integer, and *error* is a string, enclosed in double quotes, that can be printed by the command **:drc why** if the rule is violated. A width rule requires that all regions containing any types in the set *types* must be wider than *w* in both dimensions. For example, in Table 12b, the rule

width nwell 6 *''N-Well width must be at least 6 (MOSIS rule #1.1)''*

means that nwells must be at least 6 units wide whenever they appear. The *type-list* field may contain more than a single type, as in the following rule:

width allDiff 2 *''Diffusion width must be at least 2 (MOSIS rule #2.1)''*

which means that all regions consisting of the types containing any kind of diffusion be at least 2 units wide. Because many of the rules in the **drc** section refer to the same sets of layers, the **#define** facility of the C preprocessor is used to define a number of macros for these sets of layers. Table 12a gives a complete list.

All of the layers named in any one width rule must lie on the same plane. However, if some of the layers are contacts, Magic will substitute a different contact image if the named image isn't on the same plane as the other layers.

spacing	allPoly	allPoly	2	**touching_ok **
	‘‘Polysilicon spacing must be at least 2 (MOSIS rule #3.2)’’			
spacing	pfet	nncont,nnd	3	**touching_illegal **
	‘‘Transistors must be separated from substrate contacts by 3 (MOSIS rule #4.1)’’			
spacing	pc	allDiff	1	**touching_illegal **
	‘‘Poly contact must be 1 unit from diffusion (MOSIS rule #5B.6)’’			

Table 12c. Some spacing rules in the **drc** section.

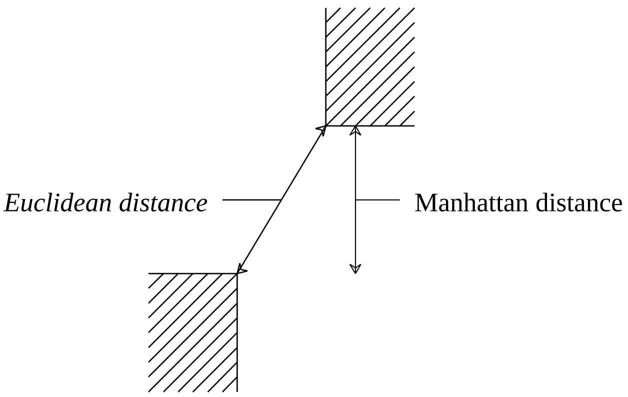


Figure 5. For design rule checking, the Manhattan distance between two horizontally or vertically aligned points is just the normal Euclidean distance. If they are not so aligned, then the Manhattan distance is the length of the longest side of the right triangle forming the diagonal line between the points.

11.2. Spacing rules

The second simple kind of design rule is a **spacing** rule. It comes in two flavors: **touching_ok**, and **touching_illegal**, both with the following syntax:

> **spacing** *types1 types2 distance flavor error*

The first flavor, **touching_ok**, does not prohibit *types1* and *types2* from being immediately adjacent. It merely requires that any type in the set *types1* must be separated by a ‘‘Manhattan’’ distance of at least *distance* units from any type in the set *types2* that is not immediately adjacent to the first type. See Figure 5 for an explanation of Manhattan distance for design rules. As an example, consider the metal1 separation rule:

> **spacing** allPoly allPoly 2 **touching_ok** \
> *‘‘Polysilicon spacing must be at least 2 (MOSIS rule #3.2)’’*

This rule is symmetric (*types1* is equal to *types2*), and requires, for example, that a pcontact be separated by at least 2 units from a piece of polysilicon. However, this rule does not prevent the pcontact from touching a piece of poly. In **touching_ok** rules, all of the layers in both *types1* and *types2* must be stored on the same plane (Magic will substitute different contact images if necessary).

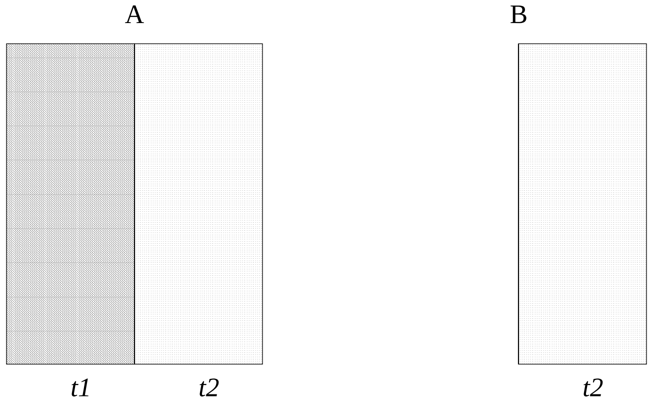


Figure 6. The **touching_ok** rules cancels spacing checks if the material is touching. This means that even distant material won't be checked for spacing. If the rule applied at edge A is a touching_ok rule between material t1 and t2, then no check will be made between the t1 material and the t2 material on the far right side of the diagram. If this check was desired, it could be accomplished in this case by a **edge4way** check from edge B. This would not work in general, though, because that check could also be masked by material of type t2, causing the touching_ok rule to be invoked.

TOUCHING_OK SPACING RULES DO NOT WORK FOR VERY LARGE SPACINGS (RELATIVE TO THE TYPES INVOLVED). SEE FIGURE 6 FOR AN EXPLANATION. If the spacing to be checked is greater than the width of one of the types involved plus either its self-spacing or spacing to a second involved type, **touching_ok spacing** may not work properly: a violation can be masked by an intervening touching type. In such cases the rule should be written using the **edge4way** construct described below.

The second flavor of spacing rule, **touching_illegal**, disallows adjacency. It is used for rules where *types1* and *types2* can never touch, as in the following:

> **spacing** pc allDiff 1 **touching_illegal** \
> *‘‘Poly contact must be 1 unit from diffusion (MOSIS rule #5B.6)’’*

Pcontacts and any type of diffusion must be at least 1 unit apart; they cannot touch. In **touching_illegal** rules *types1* and *types2* may not have any types in common: it would be rather strange not to permit a type to touch itself. In **touching_illegal** rules, *types1* and *types2* may be spread across multiple planes; Magic will find violations between material on different planes.

11.3. Edge rules

The width and spacing rules just described are actually translated by Magic into an underlying, edge-based rule format. This underlying format can handle rules more general than simple widths and spacings, and is accessible to the writer of a technology file via **edge** rules. These rules are applied at boundaries between material of two different types, in any of four directions as shown in Figure 7. The design rule table contains a separate list of rules for each possible combination of materials on the two sides of an edge.

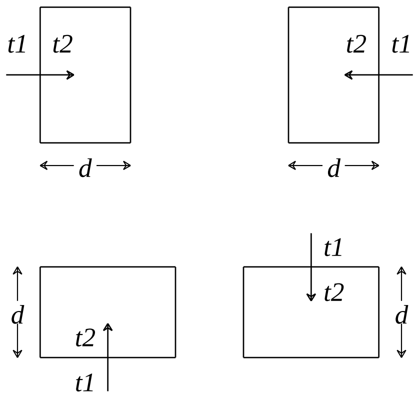


Figure 7. Design rules are applied at the edges between tiles in the same plane. A rule is specified in terms of type *t1* and type *t2*, the materials on either side of the edge. Each rule may be applied in any of four directions, as shown by the arrows. The simplest rules require that only certain mask types can appear within distance *d* on *t2*'s side of the edge.

In its simplest form, a rule specifies a distance and a set of mask types: only the given types are permitted within that distance on *type2*'s side of the edge. This area is referred to as the *constraint region*. Unfortunately, this simple scheme will miss errors in corner regions, such as the case shown in Figure 8. To eliminate these problems, the full rule format allows the constraint region to be extended past the ends of the edge under some circumstances. See Figure 9 for an illustration of the corner rules and how they work. Table 13 gives a complete description of the information in each design rule.

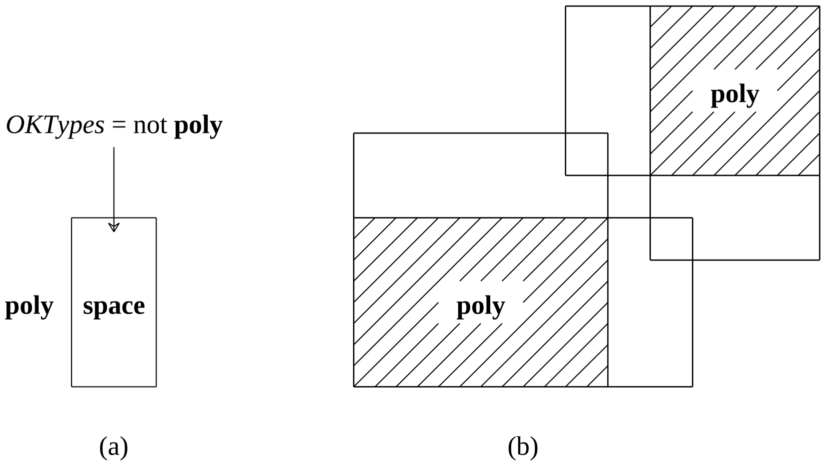


Figure 8. If only the simple rules from Figure 7 are used, errors may go unnoticed in corner regions. For example, the polysilicon spacing rule in (a) will fail to detect the error in (b).

Edge rules are specified in the technology file using the following syntax:

edge *types1 types2 d OKTypes cornerTypes cornerDist error* [*plane*]

Both *types1* and *types2* are type-lists. An edge rule is generated for each pair consisting of a type from *types1* and a type from *types2*. All the types in *types1*, *types2*, and *cornerTypes* must lie on a single plane. See Figure 9 for an example edge rule. It is sometimes useful to specify a null list, i.e., **0**, for *OKTypes* or *CornerTypes*. Null *OKTypes* means no edges between *types1* and *types2* are OK. Null *CornerTypes* means no corner extensions are to be checked (corner extensions are explained below).

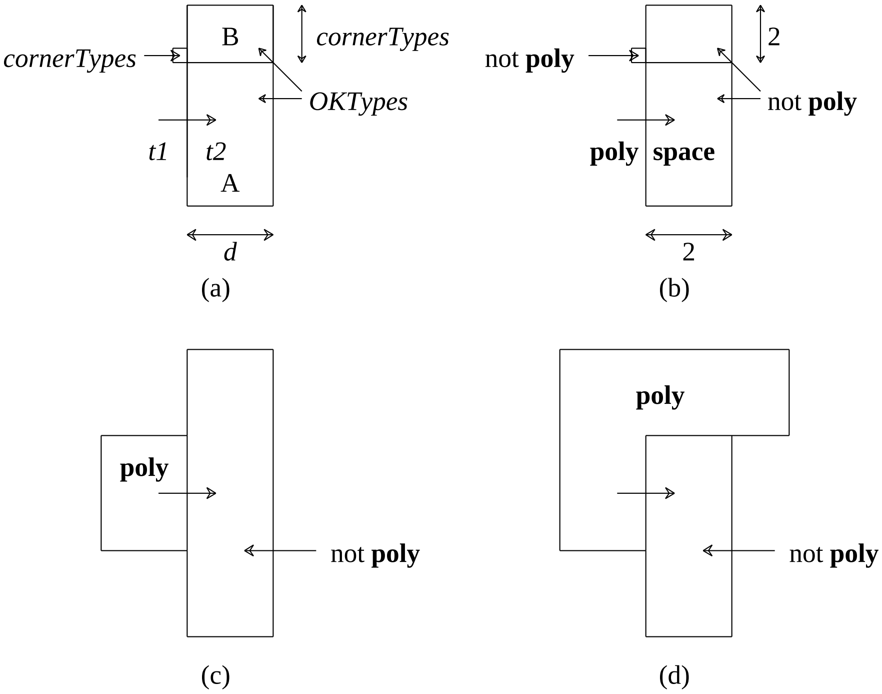


Figure 9. The complete design rule format is illustrated in (a). Whenever an edge has *type1* on its left side and *type2* on its right side, the area A is checked to be sure that only *OKTypes* are present. If the material just above and to the left of the edge is one of *cornerTypes*, then area B is also checked to be sure that it contains only *OKTypes*. A similar corner check is made at the bottom of the edge. Figure (b) shows a polysilicon spacing rule, (c) shows a situation where corner extension is performed on both ends of the edge, and (d) shows a situation where corner extension is made only at the bottom of the edge. If the rule described in (d) were to be written as an **edge** rule, it would look like:

edge poly space 2 ~poly ~poly 2 \
"Poly-poly separation must be at least 2"

Some of the edge rules in Magic have the property that if a rule is violated between two pieces of geometry, the violation can be discovered looking from either piece of geometry toward the other. To capitalize on this, Magic normally applies an edge rule only in two of the four possible directions: bottom-to-top and left-to-right, reducing the work it has to do by a factor of two. Also, the corner extension is only performed to one side of the edge: to the top for a left-to-right rule, and to the left for a bottom-to-top rule. All of the width and spacing rules translate neatly into edge rules.

However, you'll probably find it easiest when you're writing edge rules to insist that they be checked in all directions. To do this, write the rule the same way except use the keyword **edge4way** instead of **edge**:

Parameter	Meaning
type1	Material on first side of edge.
type2	Material on second side of edge.
d	Distance to check on second side of edge.
OKTypes	List of layers that are permitted within *d* units on second side of edge. (*OKTypes*=**0** means never OK)
cornerTypes	List of layers that cause corner extension. (*cornerTypes*=**0** means no corner extension)
cornerDist	Amount to extend constraint area when *cornerTypes* matches.
plane	Plane on which to check constraint region (defaults to same plane as *type1* and *type2* and *cornerTypes*).

Table 13. The parts of an edge-based rule.

edge4way nfet ndiff 2 ndiff,ndc ndiff 2 \
''Diffusion must overhang transistor by at least 2''

Not only are **edge4way** rules checked in all four directions, but the corner extension is performed on *both* sides of the edge. For example, when checking a rule from left-to-right, the corner extension is performed both to the top and to the bottom. **Edge4way** rules take twice as much time to check as **edge** rules, so it's to your advantage to use **edge** rules wherever you can.

edge4way	ppcont,ppd	ndiff,ndc,nfet	3	ndiff,ndc,nfet	ndiff,ndc,nfet	3 \
	''Ndiff must be 3 wide if it abuts ppcont or ppd (MOSIS rule #??)''					
edge4way	allPoly	~(allPoly)/active	3	~pc/active	~(allPoly)/active	3 \
	''Poly contact must be at least 3 from other poly (MOSIS rule #5B.4,5)''					
edge4way	allPoly	~(allPoly)/active	1	~m2c/metal2	~(allPoly)/active	1 \
	''Via must be on a flat surface (MOSIS rule #8.4,5)'' metal2					

Table 12d. Some edge rules in the **drc** section.

Normally, an edge rule is checked completely within a single plane: both the edge that triggers the rule and the constraint area to check fall in the same plane. However, the *plane* argument can be specified in an edge rule to force Magic to perform the constraint check on a plane different from the one containing the triggering edge. In this case, *OKTypes* must all be tile types in *plane*. This feature is used, for example, to ensure that polysilicon and diffusion edges don't lie underneath metal2 contacts:

edge4way allPoly ~(allPoly)/active 1 ~m2c/metal2 ~(allPoly)/active 1 \
''Via must be on a flat surface (MOSIS rule #8.4,5)'' metal2

11.4. Overlap Rules

In order for CIF generation and circuit extraction to work properly, certain kinds of overlaps between subcells must be prohibited. The design-rule checker provides two kinds of rules for restricting overlaps. They are

exact_overlap *type-list*
no_overlap *type-list1 type-list2*

In the **exact_overlap** rule, *type-list* indicates one or more tile types. If a cell contains a tile of one of these types and that tile is overlapped by another tile of the same type from a different cell, then the overlap must be exact: the tile in each cell must cover exactly the same area. Abutment between tiles from different cells is considered to be a partial overlap, so it is prohibited too. This rule is used to ensure that the CIF **squares** operator will work correctly, as described in Section 8.6. See Table 12e for the **exact_overlap** rule from the standard scmos technology file.

exact_overlap	m2c,ndc,pdc,pc,ppcont,nncont
no_overlap	pfet,nfet pfet,nfet

Table 12e. Exact_overlap rule in the **drc** section.

The **no_overlap** rule makes illegal any overlap between a tile in *type-list1* and a tile in *type-list2*. You should rarely, if ever, need to specify **no_overlap** rules, since Magic automatically prohibits many kinds of overlaps between subcells. After reading the technology file, Magic examines the paint table and applies the following rule: if two tile types A and B are such that the result of painting A over B is neither A nor B, or the result of painting B over A isn't the same as the result of painting A over B, then A and B are not allowed to overlap. Such overlaps are prohibited because they change the structure of the circuit. Overlaps are supposed only to connect things without making structural changes. Thus, for example, poly can overlap pcontact without violating the above rules, but poly may not overlap diffusion because the result is efet, which is neither poly nor diffusion. The only **no_overlap** rules you should need to specify are rules to keep transistors from overlapping other transistors of the same type.

11.5. Background checker step size

Magic's background design-rule checker breaks large cells up into smaller pieces, checking each piece independently. For very large designs, the number of pieces can get to be enormous. If designs are large but sparse, the performance of the design-rule checker can be improved tremendously by telling it to use a larger step size for breaking up cells. This is done as follows:

stepsize *stepsize*

which causes each cell to be processed in square pieces of at most *stepsize* by *stepsize* units. It is generally a good idea to pick a large *stepsize*, but one that is small enough so each piece will contain no more than 100 to 1000 rectangles.

12. Extract section

The **extract** section of a technology file contains the parameters used by Magic's circuit extractor. Each line in this section begins with a keyword that determines the interpretation of the remainder of the line. Table 14 gives an example **extract** section.

This section is like the **cifinput** and **cifoutput** sections in that it supports multiple extraction styles. Each style is preceded by a line of the form

style *stylename*

All subsequent lines up to the next **style** line or the end of the section are interpreted as belonging to extraction style *stylename*. If there is no initial **style** line, the first style will be named ''default''.

The keywords **areacap**, **perimcap**, and **resist** define the capacitance to substrate and the sheet resistivity of each of the Magic layers in a layout. All capacitances that appear in the **extract** section are specified as an integral number of attofarads (per unit area or perimeter), and all resistances as an integral number of milliohms per square.

The **areacap** keyword is followed by a list of types and a capacitance to substrate, as follows:

areacap *types C*

Each of the types listed in *types* has a capacitance to substrate of *C* attofarads per square lambda. Each type can appear in at most one **areacap** line. If a type does not appear in any **areacap** line, it is considered to have zero capacitance to substrate per unit area. Since most analysis tools compute transistor gate capacitance directly from the area of the transistor's gate, Magic should produce node capacitances that do not include gate capacitances. To ensure this, all transistors should have zero **areacap** values.

The **perimcap** keyword is followed by two type-lists and a capacitance to substrate, as follows:

perimcap *intypes outtypes C*

Each edge that has one of the types in *intypes* on its inside, and one of the types in *outtypes* on its outside, has a capacitance to substrate of *C* attofarads per lambda. This can also be used as an approximation of the effects due to the sidewalls of diffused areas. As for **areacap**, each unique combination of an *intype* and an *outtype* may appear at most once in a **perimcap** line. Also as for **areacap**, if a combination of *intype* and *outtype* does not appear in any **perimcap** line, its perimeter capacitance per unit length is zero.

The **resist** keyword is followed by a type-list and a resistance as follows:

resist *types R*

The sheet resistivity of each of the types in *types* is *R* milliohms per square.

Each **resist** line in fact defines a ''resistance class''. When the extractor outputs the area and perimeter of nodes in the **.ext** file, it does so for each resistance class. Normally, each resistance class consists of all types with the same resistance. However, if you wish to obtain the perimeter and area of each type separately in the **.ext** file, you should make each into its own resistance class by using a separate **resist** line for each type.

In addition to sheet resistivities, there are two other ways of specifying resistances. Neither is used by the normal Magic extractor, but both are used by the resistance extractor. Contacts have a resistance that is inversely proportional to the number of via holes in the contact, which is proportional (albeit with quantization) to the area of the contact. The **contact** keyword allows the resistance for a single via hole to be specified:

extract	
style	lambda=0.7
lambda	70
step	100
sidehalo	4
resist	poly,pfet,nfet 60000
resist	pc/a 50000
resist	pdiff,ppd 120000
resist	ndiff,nnd 120000
resist	m2contact/m1 1200
resist	metal1 200
resist	metal2,pad/m1 60
resist	ppc/a,pdc/a 100000
resist	nnc/a,ndc/a 100000
resist	nwell,pwell 3000000
areacap	poly 33
areacap	metal1 17
areacap	metal2,pad/m1 11
areacap	ndiff,nsd 350
areacap	pdiff,psd 200
areacap	ndc/a,nsc/a 367
areacap	pdc/a,psc/a 217
areacap	pcontact/a 50
perimc	allMetal1 space 56
perimc	allMetal2 space 55
overlap	metal1 pdiff,ndiff,psd,nsd 33
overlap	metal2 pdiff,ndiff,psd,nsd 17 metal1
overlap	metal1 poly 33
overlap	metal2 poly 17 metal1
overlap	metal2 metal1 33
sideoverlap	allMetal1 space allDiff 64
sideoverlap	allMetal2 space allDiff 60
sideoverlap	allMetal1 space poly 64
sideoverlap	allMetal2 space poly 60
sideoverlap	allMetal2 space allMetal1 70
fet	pfet pdiff,pdc/a 2 pfet Vdd! nwell 0 0
fet	nfet ndiff,ndc/a 2 nfet GND! pwell 0 0
end	

Table 14. **Extract** section

contact *types size R*

where *types* is a comma-separated list of types, *size* is in lambda, and *R* is in milliohms. *Size* is interpreted as a hole-size quantum; the number of holes in a contact is equal to its width divided by *size* times its length divided by *size*, with both quotients rounded down to the nearest integer. The resistance of a contact is *R* divided by the number of holes.

Transistors also have resistance information associated with them. However, a transistor's resistance may vary depending on a number of variables, so a single parameter is generally insufficient to describe it. The **fetresist** line allows sheet resistivities to be given for a variety of different configurations:

fetresist *fettypes region R*

where *fettypes* is a comma-separated list of transistor types (as defined in **fet** lines below), *region* is a string used to distinguish between resistance values for a fet if more than one is provided (the special *region* value of ‘‘**linear**’’ is required for the resistance extractor), and *R* is the on-resistance of the transistor in ohms per square (*not* milliohms; there would otherwise be too many zeroes).

Magic also extracts internodal coupling capacitances, as illustrated in Figure 10. The keywords **overlap**, **sidewall**, **sideoverlap**, and **sidehalo** provide the parameters needed to do this.

Overlap capacitance is between pairs of tile types, and is described by the **overlap** keyword as follows:

overlap *toptypes bottomtypes cap* [*shieldtypes*]

where *toptypes*, *bottomtypes*, and optionally *shieldtypes* are type-lists and *cap* is a capacitance in attofarads per square lambda. The extractor searches for tiles whose types are in *toptypes* that overlap tiles whose types are in *bottomtypes*, and that belong to different electrical nodes. (The planes of *toptypes* and *bottomtypes* must be disjoint). When such an overlap is found, the capacitance to substrate of the node of the tile in *toptypes* is deducted for the area of the overlap, and replaced by a capacitance to the node of the tile in *bottomtypes*.

If *shieldtypes* are specified, overlaps between *toptypes* and *bottomtypes* that also overlap a type in *shieldtypes* are not counted. The types in *shieldtypes* must appear on a different plane (or planes) than any of the types in *toptypes* or *bottomtypes*.

Parallel wire capacitance is between pairs of edges, and is described by the **sidewall** keyword:

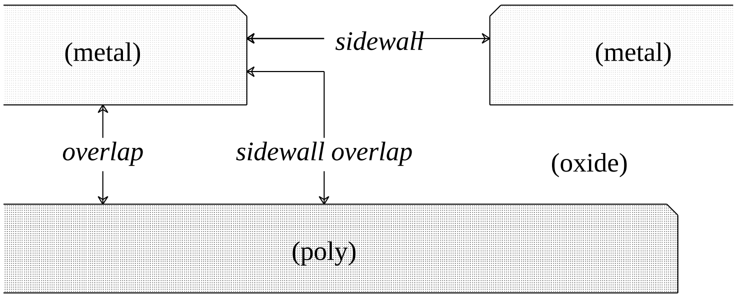


Figure 10. Magic extracts three kinds of internodal coupling capacitance. This figure is a side view of a set of masks that shows all three kinds of capacitance. *Overlap* capacitance is parallel-plate capacitance between two different kinds of material when they overlap. *Parallel wire* capacitance is fringing-field capacitance between the parallel vertical edges of two pieces of material. *Sidewall overlap* capacitance is fringing-field capacitance between the vertical edge of one piece of material and the horizontal surface of another piece of material that overlaps the vertical edge.

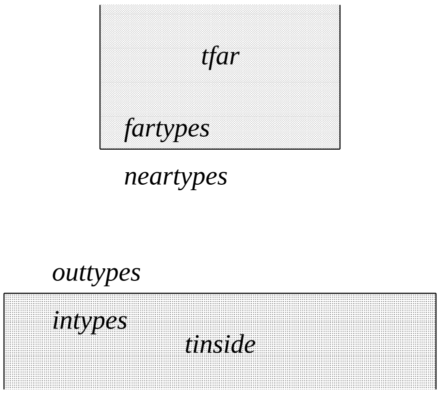


Figure 11. Parallel wire capacitance is between pairs of edges. The capacitance applies between the tiles *tinside* and *tfar* above, where *tinside*'s type is one of *intypes*, and *tfar*'s type is one of *fartypes*.

sidewall *intypes outtypes neartypes fartypes cap*

where *intypes*, *outtypes*, *neartypes*, and *fartypes* are all type-lists, described in Figure 11. *Cap* is half the capacitance in attofarads per lambda when the edges are 1 lambda apart. Parallel wire coupling capacitance is computed as being inversely proportional to the distance between two edges: at 2 lambda separation, it is equal to the value *cap*; at 4 lambda separation, it is half of *cap*. This approximation is not very good, in that it tends to overestimate the coupling capacitance between wires that are farther apart.

To reduce the amount of searching done by Magic, there is a threshold distance beyond which the effects of parallel wire coupling capacitance are ignored. This is set as follows:

sidehalo *distance*

where *distance* is the maximum distance between two edges at which Magic considers them to have parallel wire coupling capacitance. **If this number is not set explicitly in**

the technology file, it defaults to 0, with the result that no parallel wire coupling capacitance is computed.

Sidewall overlap capacitance is between material on the inside of an edge and overlapping material of a different type. It is described by the **sideoverlap** keyword:

> **sideoverlap** *intypes outtypes ovtypes cap*

where *intypes*, *outtypes*, and *ovtypes* are type-lists and *cap* is capacitance in attofarads per lambda. This is the capacitance associated with an edge with a type in *intypes* on its inside and a type in *outtypes* on its outside, that overlaps a tile whose type is in *ovtypes*. See Figure 10.

Transistors are represented in Magic by explicit tiletypes. The extraction of a fet (with gate, sources, and drains) from a collection of transistor tiles is governed by the information in a **fet** line. This line has the following format:

> **fet** *types dtypes min-nterms name snode* [*stypes*] *gscap gccap*

Types is a list of those tiletypes that make up this type of transistor. Normally, there will be only one type in this list, since Magic usually represents each type of transistor with a different tiletype.

Dtypes is a list of those tiletypes that connect to the diffusion terminals of the fet. Each transistor of this type must have at least *min-nterms* distinct diffusion terminals; otherwise, the extractor will generate an error message. For example, an **efet** in the scmos technology must have a source and drain in addition to its gate; *min-nterms* for this type of fet is 2. The tiletypes connecting to the gate of the fet are the same as those specified in the **connect** section as connecting to the fet tiletype itself.

Name is a string used to identify this type of transistor to simulation programs.

The substrate terminal of a transistor is determined in one of two ways. If *stypes* (a comma-separated list of tile types) is given, and a particular transistor overlaps one of those types, the substrate terminal will be connected to the node of the overlapped material. Otherwise, the substrate terminal will be connected to the node with the global name of *snode* (which *must* be a global name, i.e., end in an exclamation point).

Gscap is the capacitance between the transistor's gate and its diffusion terminals, in attofarads per lambda. Finally, *gccap* is the capacitance between the gate and the channel, in attofarads per square lambda. *Currently, gscap* and *gccap* are unused by the extractor.

Often the units in the extracted circuit for a cell will always be multiples of certain basic units larger than centimicrons for distance, attofarads for capacitance, or milliohms for resistance. To allow larger units to be used in the **.ext** file for this technology, thereby reducing the file's size, the **extract** section may specify a scale for any of the three units, as follows:

> **cscale** *c*
> **lambda** *l*
> **rscale** *r*

In the above, *c* is the number of attofarads per unit capacitance appearing in the **.ext** files, *l* is the number of centimicrons per unit length, and *r* is the number of milliohms per unit resistance. All three must be integers; *r* should divide evenly all the resistance-per-square values specified as part of **resist** lines, and *c* should divide evenly all the capacitance-per-unit values.

Magic's extractor breaks up large cells into chunks for hierarchical extraction, to avoid having to process too much of a cell all at once and possibly run out of memory. The size of these chunks is determined by the **step** keyword:

step *step*

This specifies that chunks of *step* units by *step* units will be processed during hierarchical extraction. The default is **100** units. Be careful about changing *step*; if it is too small then the overhead of hierarchical processing will increase, and if it is too large then more area will be processed during hierarchical extraction than necessary. It should rarely be necessary to change *step* unless the minimum feature size changes dramatically; if so, a value of about 50 times the minimum feature size appears to work fairly well.

wiring
contact pdcontact 4 metal1 0 pdiff 0
contact ndcontact 4 metal1 0 ndiff 0
contact pcontact 4 metal1 0 poly 0
contact m2contact 4 metal1 0 metal2 0
end

Table 15. **Wiring** section

13. Wiring section

The **wiring** section provides information used by the **:wire switch** command to generate contacts. See Table 15 for the **wiring** section from the scmos technology file. Each line in the section has the syntax

contact *type minSize layer1 surround1 layer2 surround2*

Type is the name of a contact layer, and *layer1* and *layer2* are the two wiring layers that it connects. *MinSize* is the minimum size of contacts of this type. If *Surround1* is non-zero, then additional material of type *layer1* will be painted for *surround1* units around contacts of *type*. If *surround2* is non-zero, it indicates an overlap distance for *layer2*.

During **:wire switch** commands, Magic scans the wiring information to find a contact whose *layer1* and *layer2* correspond to the previous and desired new wiring materials (or vice versa). If a match is found, a contact is generated according to *type*, *minSize*, *surround1*, and *surround2*.

14. Router section

The **router** section of a technology file provides information used to guide the automatic routing tools. The section contains four lines. See Table 16 for an example **router** section.

router
layer1 metal1 3 allMetal1 3
layer2 metal2 3 allMetal2 4 allPoly,allDiff 1
contacts m2contact 4
gridspacing 8
end

Table 16. **Router** section

The first two lines have the keywords **layer1** and **layer2** and the following format:

layer1 *wireType wireWidth type-list distance type-list distance ...*

They define the two layers used for routing. After the **layer1** or **layer2** keyword are two fields giving the name of the material to be used for routing that layer and the width to use for its wires. The remaining fields are used by Magic to avoid routing over existing material in the channels. Each pair of fields contains a list of types and a distance. The distance indicates how far away the given types must be from routing on that layer. Layer1 and layer2 are not symmetrical: wherever possible, Magic will try to route on layer1 in preference to layer2. Thus, in a single-metal process, metal should always be used for layer1.

The third line provides information about contacts. It has the format

contacts *contactType size* [*surround1 surround2*]

The tile type *contactType* will be used to make contacts between layer1 and layer2. Contacts will be *size* units square. In order to avoid placing contacts too close to hand-routed material, Magic assumes that both the layer1 and layer2 rules will apply to contacts. If *surround1* and *surround2* are present, they specify overlap distances around contacts for layer1 and layer2: additional layer1 material will be painted for *surround1* units around each contact, and additional layer2 material will be painted for *surround2* units around contacts.

The last line of the **routing** section indicates the size of the grid on which to route. It has the format

gridspacing *distance*

The *distance* must be chosen large enough that contacts and/or wires on adjacent grid lines will not generate any design rule violations.

plowing	
fixed	pfet,nfet,glass,pad
covered	pfet,nfet
drag	pfet,nfet
end	

Table 17. **Plowing** section

15. Plowing section

The **plowing** section of a technology file identifies those types of tiles whose sizes and shapes should not be changed as a result of plowing. Typically, these types will be transistors and buried contacts. The section currently contains three kinds of lines:

fixed *types*
covered *types*
drag *types*

where *types* is a type-list. Table 17 gives this section for the scmos technology file.

In a **fixed** line, each of *types* is considered to be fixed-size; regions consisting of tiles of these types are not deformed by plowing. Contact types are always considered to be fixed-size, so need not be included in *types*.

In a **covered** line, each of *types* will remain ''covered'' by plowing. If a face of a covered type is covered with a given type before plowing, it will remain so afterwards. For example, if a face of a transistor is covered by diffusion, the diffusion won't be allowed to slide along the transistor and expose the channel to empty space. Usually, you should make all fixed-width types covered as well, except for contacts.

In a **drag** line, whenever material of a type in *types* moves, it will drag with it any minimum-width material on its trailing side. This can be used, for example, to insure that when a transistor moves, the poly-overlap forming its gate gets dragged along in its entirety, instead of becoming elongated.

16. Plot section

The **plot** section of the technology file contains information used by Magic to generate hardcopy plots of layouts. Plots can be generated in different styles, which correspond to different printing mechanisms. For each style of printing, there is a separate subsection within the **plot** section. Each subsection is preceded by a line of the form

plot		
style	gremlin	
	poly,efet,dfet,bc,pcontact/active	18
	diff,efet,dfet,bc,ndc/active	22
	metal1,ndc/metal1,pcontact/metal1	11
	pcontact/metal1,ndc/metal1,bc	
style	versatec	
	poly,efet,dfet,bc,pcontact/active	0808 0404 0202 0101 \
		8080 4040 2020 1010 \
		0808 0404 0202 0101 \
		8080 4040 2020 1010
	diff,efet,dfet,bc,ndc/active	0000 4242 6666 0000 \
		0000 2424 6666 0000 \
		0000 4242 6666 0000 \
		0000 2424 6666 0000
	metal1,ndc/metal1,pcontact/metal1	8080 0000 0000 0000 \
		0808 0000 0000 0000 \
		8080 0000 0000 0000 \
		0808 0000 0000 0000
	pcontact/metal1,ndc/metal1,bc	X
style	colorversatec	
	poly,efet,dfet,bc,pcontact/active magenta	0808 0404 0202 0101 \
		8080 4040 2020 1010 \
		0808 0404 0202 0101 \
		8080 4040 2020 1010
	diff,efet,dfet,bc,ndc/active yellow	0000 4242 6666 0000 \
		0000 2424 6666 0000 \
		0000 4242 6666 0000 \
		0000 2424 6666 0000
	metal1,ndc/metal1,pcontact/metal1 cyan	8080 0000 0000 0000 \
		0808 0000 0000 0000 \
		8080 0000 0000 0000 \
		0808 0000 0000 0000
	pcontact/metal1,ndc/metal1,bc	X
end		

Table 17. Sample **plot** section (for an NMOS process)

style *styleName*

Right now, only **gremlin**, **versatec**, and **colorversatec** styles are supported.

Within the **gremlin** subsection, lines must have one of three forms:

type-list stippleNumber
type-list **X**
type-list **B**

The first form of line associates a Gremlin stipple number with all Magic layers in *type-list*. When Gremlin files are generated, all areas covered by *type-list* will appear as stippled areas filled with stipple *stippleNumber* and bordered with thin solid lines. The second form is designed for contacts. It causes each tile in *type-list* to be outlined with a medium-thickness line with an additional medium-thickness ''X'' drawn between opposite corners. The **B** specification is identical to **X** except that only the border is drawn, without the diagonal ''X''.

Within the **versatec** subsection, lines may also be in either of three forms:

type-list pat0 pat1 ... pat15
type-list **X**
type-list **B**

In the first case, the material of types *type-list* is rendered with a stipple pattern given by 16 hexadecimal numbers. Each number contains four hex digits; the result is a 16-by-16 bit pattern of 1's and 0's. A one means that the corresponding bit of the output file is set and a zero means that the bit is not modified when this layer is rendered (thus the patterns from different *type-list*s will OR together). *Pat0* specifies the top line of the stipple pattern; within each pattern, the most significant bit corresponds to the leftmost bit within the line of the stipple pattern. Stippled areas area also bordered by thin solid lines. The second and third forms (**X** and **B**) are similar to the second and third forms for **gremlin** lines: Magic outlines tiles in *type-list* with medium-thickness lines and also draws crosses through the tiles if **X** is given.

The **colorversatec** subsection is just like the **versatec** section except the stipple lines can also specify a color:

type-list color pat0 pat1 ... pat15

where *color* is one of **black**, **cyan**, **magenta**, or **yellow**. This color is the dye that will be used for the stipple in the plot. Multicolored stipples may be obtained by listing the same *type-list* and stipple patterns two or more times, each with a different *color*.

For **versatec** plotting there are a number of parameters that can be set directly by users, such as the printer width. These parameters allow users to reconfigure the system for different kinds of plotters and different spooling mechanisms. See the manual page for details. You may want to modify your system **.magic** file to set up default parameters for your printer.

17. Installing a Technology File

As mentioned earlier, ''raw'' technology files cannot be read directly by Magic. The C preprocessor must first be used to eliminate comments and expand macros in a technology file before it gets installed. As a consequence, the full power of the C preprocessor is available to the writer of a technology file. Not only may macro definitions be made with **#define**, but ''conditional compilation'' using **#ifdef** and the ability to use other files via the **#include** mechanism are possible.

Technology files are installed as a file of the name *techname***.tech***n*. The numeric version suffix *n* (currently **26**) is added to the final **.tech** when the file is installed, and allows multiple versions of the technology file to coexist in the same directory. There is a shell script, **tech/:techinstall**, to do all the necessary processing to install a new technology file.

Technology files can be installed in any directory. When Magic is run, it searches for a technology file first in the current directory and next in the system library directory, ~**cad/lib/magic/sys**. To install a new technology file whose source is *techname***.tech**, run:

tech/:techinstall *techname***.tech** *vers dir*

where *dir* is the directory in which the technology file is to be installed, and *vers* is the proper version suffix to insure that this technology file is readable by the latest version of Magic. See the Makefile in **tech** for the string **VERSION**, which defines the current version number.