Magic Tutorial #1: Getting Started

John Ousterhout

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

1 What is Magic?

Magic is an interactive system for creating and modifying VLSI circuit layouts. With Magic, you use a color graphics display and a mouse or graphics tablet to design basic cells and to combine them hierarchically into large structures. Magic is different from other layout editors you may have used. The most important difference is that Magic is more than just a color painting tool: it understands quite a bit about the nature of circuits and uses this information to provide you with additional operations. For example, Magic has built-in knowledge of layout rules; as you are editing, it continuously checks for rule violations. Magic also knows about connectivity and transistors, and contains a built-in hierarchical circuit extractor. Magic also has a *plow* operation that you can use to stretch or compact cells. Lastly, Magic has routing tools that you can use to make the global interconnections in your circuits.

Magic is based on the Mead-Conway style of design. This means that it uses simplified design rules and circuit structures. The simplifications make it easier for you to design circuits and permit Magic to provide powerful assistance that would not be possible otherwise. However, they result in slightly less dense circuits than you could get with more complex rules and structures. For example, Magic permits only *Manhattan* designs (those whose edges are vertical or horizontal). Circuit designers tell us that our conservative design rules cost 5-10% in density. We think that the density sacrifice is compensated for by reduced design time.

2 How to Get Help and Report Problems

There are several ways you can get help about Magic. If you are trying to learn about the system, you should start off with the Magic tutorials, of which this is the first. Each tutorial introduces a

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #3: Advanced Painting (Wiring and Plowing)
Magic Tutorial #4: Cell Hierarchies
Magic Tutorial #5: Multiple Windows
Magic Tutorial #6: Design-Rule Checking
Magic Tutorial #7: Netlists and Routing
Magic Tutorial #8: Circuit Extraction
Magic Tutorial #9: Format Conversion for CIF and Calma
Magic Tutorial #10: The Interactive Route
Magic Tutorial #11: Using RSIM with Magic
Magic Maintainer's Manual #1: Hints for System Maintainers
Magic Maintainer's Manual #2: The Technology File
Magic Maintainer's Manual #3: Display Styles, Color Maps, and Glyphs
Magic Maintainer's Manual #4: Using Magic Under X Windows
Magic Technology Manual #1: NMOS
Magic Technology Manual #2: SCMOS

Table 1: The Magic tutorials, maintenance manuals, and technology manuals.

particular set of facilities in Magic. There is also a set of manuals intended for system maintainers. These describe things like how to create new technologies. Finally, there is a set of technology manuals. Each one of the technology manuals describes the features peculiar to a particular technology, such as layer names and design rules. Table 1 lists all of the Magic manuals. The tutorials are designed to be read while you are running Magic, so that you can try out the new commands as they are explained. You needn't read all the tutorials at once; each tutorial lists the other tutorials that you should read first.

The tutorials are not necessarily complete. Each one is designed to introduce a set of facilities, but it doesn't necessarily cover every possibility. The ultimate authority on how Magic works is the reference manual, which is a standard Unix *man* page. The *man* page gives concise and complete descriptions of all the Magic commands. Once you have a general idea how a command works, the *man* page is probably easier to consult than the tutorial. However, the *man* page may not make much sense until after you've read the tutorial.

A third way of getting help is available on-line through Magic itself. The **:help** command will print out one line for each Magic command, giving the command's syntax and an extremely brief description of the command. This facility is useful if you've forgotten the name or exact syntax of a command. After each screenful of help information, **:help** stops and prints "–More–". If you type a space, the next screenful of data will be output, and if you type **q** the rest of the output will be skipped. If you're interested in information about a particular subject, you can type

:help *subject*

This command will print out each command description that contains the *subject* string.

If you have a question or problem that can't be answered with any of the above approaches, you may contact the Magic authors by sending mail to `magic@ucbarpa.Berkeley.EDU`.

This will log your message in a file (so we can't forget about it) and forward the message to the Magic maintainers. Magic maintenance is a mostly volunteer effort, so when you report a bug or ask a question, *please* be specific. Obviously, the more specific you are, the more likely we can answer your question or reproduce the bug you found. We'll tend to answer the specific bug reports first, since they involve less time on our part. Try to describe the exact sequence of events that led to the problem, what you expected to happen, and what actually happened. If possible, find a small example that reproduces the problem and send us the relevant (small!) files so we can make it happen here. Or best of all, send us a bug fix along with a small example of the problem.

3 Graphics Configuration

Magic can be run with different graphics hardware. The most common configuration is to run Magic under X11 on a workstation. Another way to run Magic is under SunView on a Sun workstation, or under OpenGL (in an X11 environment) on an SGI workstation or Linux box with accelerated 3D video hardware and drivers. Legacy code exists supporting AED graphics terminals and X10 (the forerunner of X11). The rest of this section concerns X11.

Before starting up magic, make sure that your `DISPLAY` variable is set correctly. If you are running magic and your X server on the same machine, set it to `unix:0`:

```
setenv DISPLAY unix:0
```

The Magic window is an ordinary X window, and can be moved and resized using the window manager.

For now, you can skip to the next major section: ”Running Magic”.

4 Advanced X Use

The X11 driver can read in window sizing and font preferences from your *.Xdefaults* file. The following specifications are recognized:

magic.window:	1000x600+10+10
magic.newwindow:	300x300+400+100
magic.small:	helvetica8
magic.medium:	helvetica12
magic.large:	helvetica18
magic.xlarge:	helvetica24

magic.window is the size and position of the initial window, while **magic.newwindow** is the size and position of subsequent windows. If these are left blank, you will be prompted to give the window's position and size. **small**, **medium**, **large**, and **xlarge** are various fonts magic uses for labels. Some X11 servers read the `.Xdefaults` file only when you initially log in; you may have to run `xrdb -load ~/.Xdefaults` for the changes to take effect.

Under X11, Magic can run on a display of any depth for which there are colormap and dstyle files. Monochrome, 4 bit, 6 bit, 7 bit, and 24 bit files for MOS are distributed in this release. You can explicitly specify how many planes Magic is to use by adding a suffix numeral between 1 and 7 to "XWIND" when used with Magic's "-d" option. For example, "magic -d XWIND1" runs magic on a monochrome display and "magic -d XWIND7" runs magic on a 7 plane display. If this number is not specified, magic checks the depth of the display and picks the largest number in the set {1, 4, 6, 7, 16, 24} that the display will support. Another way to force the display type is to set an environment variable called `MAGIC_COLOR` to one of the strings "8bit", "16bit", or "24bit".

Linux note:
Magic's "native" display (except when using the OpenGL interface) is the 8-bit PseudoColor visual type. 24-bit TrueColor visuals prevent Magic from allocating colors for bit-plane logical operations, so the 24-bit interface is visually somewhat sub-par, requiring stipple patterns on all metal layers, for instance. Under Linux, a few (commercial) X drivers will support 8-bit overlays on top of 24-bit TrueColor when using 32-bit color. This is the ideal way to use magic, because the colormap for the rest of the display is preserved when the cursor is inside the Magic window. Otherwise, the X session may have to be started using "`startx --bpp 8`" to force it to use the 8-bit PseudoColor visual.

X11 remote usage note:
When running Magic remotely on an X terminal, the colormap allocation may differ for the local machine compared to the remote machine. In some cases, this can cause the background of magic to appear black, usually with a black-on-black cursor. This is known to be true of X11 drivers for Windows (such as PC-XWare), due to the way the Windows 8-bit PseudoColor colormap is set up. This behavior can be corrected by setting two environment variables on the remote machine as follows:

```
setenv X_COLORMAP_BASE 128
setenv X_COLORMAP_DEFAULT 0
```

This causes Magic to avoid trying to allocate the first color in the colormap, which under Windows is fixed as black.

5 Running Magic

From this point on, you should be sitting at a Magic workstation so you can experiment with the program as you read the manuals. Starting up Magic is usually pretty simple. Just log in and, if needed, start up your favorite window system. Then type the shell command

magic tut1

Tut1 is the name of a library cell that you will play with in this tutorial. At this point, several colored rectangles should appear on the color display along with a white box and a cursor. A message will be printed on the text display to tell you that **tut1** isn't writable (it's in a read-only library), and a ">" prompt should appear. If this has happened, then you can skip the rest of this section (except for the note below) and go directly to Section 5.

Note: in the tutorials, when you see things printed in boldface, for example, **magic tut1** from above, they refer to things you type exactly, such as command names and file names. These are usually case sensitive (**A** is different from **a**). When you see things printed in italics, they refer to classes of things you might type. Arguments in square brackets are optional. For example, a more complete description of the shell command for Magic is

magic [*file*]

You could type any file name for *file*, and Magic would start editing that file. It turns out that **tut1** is just a file in Magic's cell library. If you didn't type a file name, Magic would load a new blank cell.

If things didn't happen as they should have when you tried to run Magic, any of several things could be wrong. If a message of the form "magic: Command not found" appears on your screen it is because the shell couldn't find the Magic program. The most stable version of Magic is the directory `~cad/bin`, and the newest public version is in `~cad/new`. You should make sure that both these directories are in your shell path. Normally, `~cad/new` should appear before `~cad/bin`. If this sounds like gibberish, find a Unix hacker and have him or her explain to you about paths. If worst comes to worst, you can invoke Magic by typing its full name:

~cad/bin/magic tut1

Another possible problem is that Magic might not know what kind of display you are using. To solve this, use magic's **-d** flag:

magic -d *display* **tut1**

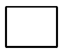
Display is usually the model number of the workstation you are using or the name of your window system. Look in the manual page for a list of valid names, or just guess something. Magic will print out the list of valid names if you guess wrong.

If you are using a graphics terminal (not a workstation), it is possible that Magic doesn't know which serial line to use. To learn how to fix this, read about the **-g** switch in the magic(1) manual page. Also read the displays(5) manual page.

6 The Box and the Cursor

Two things, called the *box* and the *cursor*, are used to select things on the color display. As you move the mouse, the cursor moves on the screen. The cursor starts out with a crosshair shape, but you'll see later that its shape changes as you work to provide feedback about what you're doing. The left and right mouse buttons are used to position the box. If you press the left mouse button and then release it, the box will move so that its lower left corner is at the cursor position. If you press and release the right mouse button, the upper right corner of the box will move to the cursor position, but the lower left corner will not change. These two buttons are enough to position the box anywhere on the screen. Try using the buttons to place the box around each of the colored rectangles on the screen.

Sometimes it is convenient to move the box by a corner other than the lower left. To do this, press the left mouse button and *hold it down*. The cursor shape changes to show you that you are moving the box by its lower left corner:



While holding the button down, move the cursor near the lower right corner of the box, and now click the right mouse button (i.e. press and release it, while still holding down the left button). The cursor's shape will change to indicate that you are now moving the box by its lower right corner. Move the cursor to a different place on the screen and release the left button. The box should move so that its lower right corner is at the cursor position. Try using this feature to move the box so that it is almost entirely off-screen to the left. Try moving the box by each of its corners.

You can also reshape the box by corners other than the upper right. To do this, press the right mouse button and hold it down. The cursor shape shows you that you are reshaping the box by its upper right corner:

Now move the cursor near some other corner of the box and click the left button, all the while holding the right button down. The cursor shape will change to show you that now you are reshaping the box by a different corner. When you release the right button, the box will reshape so that the selected corner is at the cursor position but the diagonally opposite corner is unchanged. Try reshaping the box by each of its corners.

7 Invoking Commands

Commands can be invoked in Magic in three ways: by pressing buttons on the mouse; by typing single keystrokes on the text keyboard (these are called *macros*); or by typing longer commands on the text keyboard (these are called *long commands*). Many of the commands use the box and cursor to help guide the command.

To see how commands can be invoked from the buttons, first position the box over a small blank area in the middle of the screen. Then move the cursor over the red rectangle and press the middle mouse button. At this point, the area of the box should get painted red. Now move the cursor over empty space and press the middle button again. The red paint should go away. Note how this command uses both the cursor and box locations to control what happens.

As an example of a macro, type the **g** key on the text keyboard. A grid will appear on the color display, along with a small black box marking the origin of the cell. If you type **g** again, the grid will go away. You may have noticed earlier that the box corners didn't move to the exact cursor position: you can see now that the box is forced to fall on grid points.

Long commands are invoked by typing a colon (":") or semi-colon (";"). After you type the colon or semi-colon, the ">" prompt on the text screen will be replaced by a ":" prompt. This indicates that Magic is waiting for a long command. At this point you should type a line of text, followed by a return. When the long command has been processed, the ">" prompt reappears on the text display. Try typing semi-colon followed by return to see how this works. Occasionally a "]" (right bracket) prompt will appear. This means that the design-rule checker is reverifying part of your design. For now you can just ignore this and treat "]" like ">".

Each long command consists of the name of the command followed by arguments, if any are needed by that command. The command name can be abbreviated, just as long as you type enough characters to distinguish it from all other long commands. For example, **:h** and **:he** may be used as abbreviations for **:help**. On the other hand, **:u** may not be used as an abbreviation for **:undo** because there is another command, **:upsidedown**, that has the same abbreviation. Try typing **:u**.

As an example of a long command, put the box over empty space on the color display, then invoke the long command

:paint red

The box should fill with the red color, just as if you had used the middle mouse button to paint it. Everything you can do in Magic can be invoked with a long command. It turns out that the macros are just conveniences that are expanded into long commands and executed. For example, the long command equivalent to the **g** macro is

:grid

Magic permits you to define new macros if you wish. Once you've become familiar with Magic you'll almost certainly want to add your own macros so that you can invoke quickly the commands you use most frequently. See the *magic(1)* man page under the command **:macro**.

One more long command is of immediate use to you. It is

:quit

Invoke this command. Note that before exiting, Magic will give you one last chance to save the information that you've modified. Type **y** to exit without saving anything.

Magic Tutorial #2: Basic Painting and Selection

John Ousterhout

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started

Commands introduced in this tutorial:

:box, :clockwise, :copy, :erase, :findbox :grid, :label, :layers, :macro, :move, :paint, :redo, :save, :select, :sideways, :undo, :upsidedown, :view, :what, :writeall, :zoom

Macros introduced in this tutorial:

a, A, c, d, ˆD, e, E, g, G, q, Q, r, R, s, S, t, T, u, U, v, w, W, z, Z, 4

1 Cells and Paint

In Magic, a circuit layout is a hierarchical collection of *cells*. Each cell contains three things: colored shapes, called *paint*, that define the circuit's structure; textual *labels* attached to the paint; and *subcells*, which are instances of other cells. The paint is what determines the eventual function of the VLSI circuit. Labels and subcells are a convenience for you in managing the layout and provide a way of communicating information between various synthesis and analysis tools. This tutorial explains how to create and edit paint and labels in simple single-cell designs, using the basic painting commands. "Magic Tutorial #3: Advanced Painting (Wiring and Plowing)" describes some more advanced features for manipulating paint. For information on how to build up cell hierarchies, see "Magic Tutorial #4: Cell Hierarchies".

2 Painting and Erasing

Enter Magic to edit the cell **tut2a** (type **magic tut2a** to the Unix shell; follow the directions in "Tutorial #1: Getting Started" if you have any problems with this). The **tut2a** cell is a sort of palette: it shows a splotch of each of several paint layers and gives the names that Magic uses for the layers.

The two basic layout operations are painting and erasing. They can be invoked using the **:paint** and **:erase** long commands, or using the buttons. The easiest way to paint and erase is with the mouse buttons. To paint, position the box over the area you'd like to paint, then move the cursor over a color and click the middle mouse button. To erase everything in an area, place the box over the area, move the cursor over a blank spot, and click the middle mouse button. Try painting and erasing various colors. If the screen gets totally messed up, you can always exit Magic and restart it. While you're painting, white dots may occasionally appear and disappear. These are design rule violations detected by Magic, and will be explained in "Magic Tutorial #6: Design Rule Checking". You can ignore them for now.

It's completely legal to paint one layer on top of another. When this happens, one of three things may occur. In some cases, the layers are independent, so what you'll see is a combination of the two, as if each were a transparent colored foil. This happens, for example, if you paint metal1 (blue) on top of polysilicon (red). In other cases, when you paint one layer on top of another you'll get something different from either of the two original layers. For example, painting poly on top of ndiff produces ntransistor (try this). In still other cases the new layer replaces the old one: this happens, for example, if you paint a pcontact on top of ntransistor. Try painting different layers on top of each other to see what happens. The meaning of the various layers is discussed in more detail in Section 11 below.

There is a second way of erasing paint that allows you to erase some layers without affecting others. This is the macro **ˆD** (control-D, for "**D**elete paint"). To use it, position the box over the area to be erased, then move the crosshair over a splotch of paint containing the layer(s) you'd like to erase. Type **ˆD** key on the text keyboard: the colors underneath the cursor will be erased from the area underneath the box, but no other layers will be affected. Experiment around with the **ˆD** macro to try different combinations of paints and erases. If the cursor is over empty space then the **ˆD** macro is equivalent to the middle mouse button: it erases everything.

You can also paint and erase using the long commands

:paint *layers*
:erase *layers*

In each of these commands *layers* is one or more layer names separated by commas (you can also use spaces for separators, but only if you enclose the entire list in double-quotes). Any layer can be abbreviated as long as the abbreviation is unambiguous. For example, **:paint poly,metal1** will paint the polysilicon and metal1 layers. The macro **ˆD** is predefined by Magic to be **:erase $** (**$** is a pseudo-layer that means "all layers underneath the cursor").

3 Undo

There are probably going to be times when you'll do things that you'll later wish you hadn't. Fortunately, Magic has an undo facility that you can use to restore things after you've made mistakes. The command

:undo

(or, alternatively, the macro **u**) will undo the effects of the last command you invoked. If you made a mistake several commands back, you can type **:undo** several times to undo successive commands. However, there is a limit to all this: Magic only remembers how to undo the last ten or so commands. If you undo something and then decide you wanted it after all, you can undo the undo with the command

:redo

(**U** is a macro for this command). Try making a few paints and erases, then use **:undo** and **:redo** to work backwards and forwards through the changes you made.

4 The Selection

Once you have painted a piece of layout, there are several commands you can invoke to modify the layout. Many of them are based on the *selection*: you select one or more pieces of the design, and then perform operations such as copying, deletion, and rotation on the selected things. To see how the selection works, load cell **tut2b**. You can do this by typing **:load tut2b** if you're still in Magic, or by starting up Magic with the shell command **magic tut2b**.

The first thing to do is to learn how to select. Move the cursor over the upper portion of the L-shaped blue area in **tut2b**, and type **s**, which is a macro for **:select**. The box will jump over to cover the vertical part of the "L". This operation selected a chunk of material. Move the box away from the chunk, and you'll see that a thin white outline is left around the chunk to show that it's selected. Now move the cursor over the vertical red bar on the right of the cell and type **s**. The box will move over that bar, and the selection highlighting will disappear from the blue area.

If you type **s** several times without moving the cursor, each command selects a slightly larger piece of material. Move the cursor back over the top of the blue "L", and type **s** three times without moving the cursor. The first **s** selects a chunk (a rectangular region all of the same type of material). The second **s** selects a *region* (all of the blue material in the region underneath the cursor, rectangular or not). The third **s** selects a *net* (all of the material that is electrically connected to the original chunk; this includes the blue metal, the red polysilicon, and the contact that connects them).

The macro **S** (short for **:select more**) is just like **s** except that it adds on to the selection, rather than replacing it. Move the cursor over the vertical red bar on the right and type **S** to see how this works. You can also type **S** multiple times to add regions and nets to the selection.

If you accidentally type **s** or **S** when the cursor is over space, you'll select a cell (**tut2b** in this case). You can just undo this for now. Cell selection will be discussed in "Magic Tutorial #4: Cell Hierarchies".

You can also select material by area: place the box around the material you'd like to select and type **a** (short for **:select area**). This will select all of the material underneath the box. You can use the macro **A** to add material to the selection by area, and you can use the long command

:select [**more**]**area** *layers*

to select only material on certain layers. Place the box around everything in **tut2b** and type **:select area metal1** followed by **:select more area poly**.

If you'd like to clear out the selection without modifying any of the selected material, you can use the command

:select clear

or type the macro **C**. You can clear out just a portion of the selection by typing **:select less** or **:select less area** *layers*; the former deselects paint in the order that **:select** selects paint, while the latter deselects paint under the box (just as **:select area** selects paint under the box). For a synopsis of all the options to the **:select** command, type

:select help

5 Operations on the Selection

Once you've made a selection, there are a number of operations you can perform on it:

:delete
:move [*direction* [*distance*]]
:stretch [*direction* [*distance*]]
:copy
:upsidedown
:sideways
:clockwise [*degrees*]

The **:delete** command deletes everything that's selected. Watch out: **:delete** is different from **:erase**, which erases paint from the area underneath the box. Select the red bar on the right in **tut2b** and type **d**, which is a macro for **:delete**. Undo the deletion with the **u** macro.

The **:move** command picks up both the box and the selection and moves them so that the lower-left corner of the box is at the cursor location. Select the red bar on the right and move it so that it falls on top of the vertical part of the blue "L". You can use **t** ("**t**ranslate") as a macro for **:move**. Practice moving various things around the screen. The command **:copy** and its macro **c** are just like **:move** except that a copy of the selection is left behind at the original position.

There is also a longer form of the **:move** command that you can use to move the selection a precise amount. For example, **:move up 10** will move the selection (and the box) up 10 units. The *direction* argument can be any direction like **left**, **south**, **down**, etc. See the Magic manual page for a complete list of the legal directions. The macros **q**, **w**, **e**, and **r** are defined to move the selection left, down, up, and right (respectively) by one unit.

The **:stretch** command is similar to **:move** except that it stretches and erases as it moves. **:stretch** does not operate diagonally, so if you use the cursor to indicate where to stretch to, Magic will either stretch up, down, left, or right, whichever is closest. The **:stretch** command moves the selection and also does two additional things. First, for each piece of paint that moves, **:stretch** will erase that layer from the region that the paint passes through as it moves, in order to clear material out of its way. Second, if the back edge of a piece of selected paint touches non-selected material, one of the two pieces of paint is stretched to maintain the connection. The macros **Q**, **W**, **E**, and **R** just like the macros **q**, etc. described above for **:move**. The macro **T** is predefined to **:stretch**. To see how stretching works, select the horizontal piece of the green wire in **tut2b** and type **W**, then **E**. Stretching only worries about material in front of and behind the selection; it ignores material to the sides (try the **Q** and **R** macros to see). You can use plowing (described in Tutorial #3) if this is a problem.

The command **:upsidedown** will flip the selection upside down, and **:sideways** flips the selection sideways. Both commands leave the selection so it occupies the same total area as before, but with the contents flipped. The command **:clockwise** will rotate the selection clockwise, leaving the lower-left corner of the new selection at the same place as the lower-left corner of the old selection. *Degrees* must be a multiple of 90, and defaults to 90.

At this point you know enough to do quite a bit of damage to the **tut2b** cell. Experiment with the selection commands. Remember that you can use **:undo** to back out of trouble.

6 Labels

Labels are pieces of text attached to the paint of a cell. They are used to provide information to other tools that will process the circuit. Most labels are node names: they provide an easy way of referring to nodes in tools such as routers, simulators, and timing analyzers. Labels may also be used for other purposes: for example, some labels are treated as *attributes* that give Crystal, the timing analyzer, information about the direction of signal flow through transistors.

Load the cell **tut2c** and place a cross in the middle of the red chunk (to make a cross, position the lower-left corner of the box with the left button and then click the right button to place the upper-right corner on top of the lower-left corner). Then type type the command **:label test**. A new label will appear at the position of the box. The complete syntax of the **:label** command is

:label [*text* [*position* [*layer*]]]

Text must be supplied, but the other arguments can be defaulted. If *text* has any spaces in it, then it must be enclosed in double quotes. *Position* tells where the text should be displayed, relative to the point of the label. It may be any of **north**, **south**, **east**, **west**, **top**, **bottom**, **left**, **right**, **up**, **down**, **center**, **northeast**, **ne**, **southeast**, **se**, **southwest**, **sw**, **northwest**, **nw**. For example, if **ne** is given, the text will be displayed above and to the right of the label point. If no *position* is given, Magic will pick a position for you. *Layer* tells which paint layer to attach the label to. If *layer* covers the entire area of the label, then the label will be associated with the particular layer. If *layer* is omitted, or if it doesn't cover the label's area, Magic initially associates the label with the "space" layer, then checks to see if there's a layer that covers the whole area. If there is, Magic moves the label to that layer. It is generally a bad idea to place labels at points where there are several paint layers, since it will be hard to tell which layer the label is attached to. As you edit, Magic will ensure that labels are only attached to layers that exist everywhere under the label. To see how this works, paint the layer pdiff (brown) over the label you just created: the label will switch layers. Finally, erase poly over the area, and the label will move again.

Although many labels are point labels, this need not be the case. You can label any rectangular area by setting the box to that area before invoking the label command. This feature is used for labelling terminals for the router (see below), and for labelling tiles used by Mpack, the tile packing program. **Tut2c** has examples of point, line, and rectangular labels.

All of the selection commands apply to labels as well as paint. Whenever you select paint, the labels attached to that paint will also be selected. Selected labels are highlighted in white. Select some of the chunks of paint in **tut2c** to see how the labels are selected too. When you use area selection, labels will only be selected if they are completely contained in the area being selected. If you'd like to select *just* a label without any paint, make the box into a cross and put the cross on the label: **s** and **S** will select just the label.

There are several ways to erase a label. One way is to select and then delete it. Another way is to erase the paint that the label is attached to. If the paint is erased all around the label, then Magic will delete the label too. Try attaching a label to a red area, then paint blue over the red. If you erase blue the label stays (since it's attached to red), but if you erase the red then the label is deleted.

You can also erase labels using the **:erase** command and the pseudo-layer **labels**. The command

:erase labels

will erase all labels that lie completely within the area of the box. Finally, you can erase a label by making the box into a cross on top of the label, then clicking the middle button with the cursor over empty space. Technically, this will erase all paint layers and labels too. However, since the box has zero area, erasing paint has no effect: only the labels are erased.

7 Labelling Conventions

When creating labels, Magic will permit you to use absolutely any text whatsoever. However, many other tools, and even parts of Magic, expect label names to observe certain conventions. Except for the special cases described below, labels shouldn't contain any of the characters "/$@!^". Spaces, control characters, or parentheses within labels are probably a bad idea too. Many of the programs that process Magic output have their own restrictions on label names, so you should find out about the restrictions that apply at your site. Most labels are node names: each one gives a unique identification to a set of things that are electrically connected. There are two kinds of node names, local and global. Any label that ends in "!" is treated as a global node name; it will be assumed that all nodes by this name, anywere in any cell in a layout, are electrically connected. The most common global names are **Vdd!** and **GND!**, the power rails. You should always use these names exactly, since many other tools require them. Nobody knows why "GND!" is all in capital letters and "Vdd!" isn't.

Any label that does not end in "!" or any of the other special characters discussed below is a local node name. It refers to a node within that particular cell. Local node names should be unique within the cell: there shouldn't be two electrically distinct nodes with the same name. On the other hand, it is perfectly legal, and sometimes advantageous, to give more than one name to the same node. It is also legal to use the same local node name in different cells: the tools will be able to distinguish between them and will not assume that they are electrically connected.

The only other labels currently understood by the tools are *attributes*. Attributes are pieces of text associated with a particular piece of the circuit: they are not node names, and need not be unique. For example, an attribute might identify a node as a chip input, or it might identify a transistor terminal as the source of information for that transistor. Any label whose last character is "@", "$", or "ˆ" is an attribute. There are three different kinds of attributes. Node attributes are those ending with "@"; they are associated with particular nodes. Transistor source/drain attributes are those ending in "$"; they are associated with particular terminals of a transistor. A source or drain attribute must be attached to the channel region of the transistor and must fall exactly on the source or drain edge of the transistor. The third kind of attribute is a transistor gate attribute. It ends in "ˆ" and is attached to the channel region of the transistor. To see examples of attributes and node names, edit the cell **tut2d** in Magic.

Special conventions apply to labels for routing terminals. The standard Magic router (invoked by **:route**) ignores all labels except for those on the edges of cells. (This restriction does not apply to the gate-array router, Garoute, or to the interactive router, Iroute). If you expect to use the standard router to connect to a particular node, you should place the label for that node on its outermost edge. The label should not be a point label, but should instead be a horizontal or vertical line covering the entire edge of the wire. The router will choose a connection point somewhere along the label. A good rule of thumb is to label all nodes that enter or leave the cell in this way. For more details on how labels are used by the standard router, see "Magic Tutorial #7: Netlists and Routing". Other labeling conventions are used by the Garouter and Irouter, consult their respective tutorials for details.

8 Files and Formats

Magic provides a variety of ways to save your cells on disk. Normally, things are saved in a special Magic format. Each cell is a separate file, and the name of the file is just the name of the cell with **.mag** appended. For example, the cell **tut2a** is saved in file **tut2a.mag**. To save cells on disk, invoke the command

```
:writeall
```

This command will run through each of the cells that you have modified in this editing session, and ask you what to do with the cell. Normally, you'll type **write**, or just hit the return key, in which case the cell will be written back to the disk file from which it was read (if this is a new cell, then you'll be asked for a name for the cell). If you type **autowrite**, then Magic will write out all the cells that have changed without asking you what to do on a cell-by-cell basis. **Flush** will cause Magic to delete its internal copy of the cell and reload the cell from the disk copy, thereby expunging all edits that you've made. **Skip** will pass on to the next cell without writing this cell (but Magic still remembers that it has changed, so the next time you invoke **:writeall** Magic will ask about this cell again). **Abort** will stop the command immediately without writing or checking any more cells.

IMPORTANT NOTE: Unlike vi and other text editors, Magic doesn't keep checkpoint files. This means that if the system should crash in the middle of a session, you'll lose all changes since the last time you wrote out cells. It's a good idea to save your cells frequently during long editing sessions.

You can also save the cell you're currently editing with the command

:save *name*

This command will append ".mag" to *name* and save the cell you are editing in that location. If you don't provide a name, Magic will use the cell's name (plus the ".mag" extension) as the file name, and it will prompt you for a name if the cell hasn't yet been named.

Once a cell has been saved on disk you can edit it by invoking Magic with the command

magic *name*

where *name* is the same name you used to save the cell (no ".mag" extension).

Magic can also read and write files in CIF and Calma Stream formats. See "Magic Tutorial #9: Format Conversion for CIF and Calma" for details.

9 Plotting

Magic can generate hardcopy plots of layouts in four ways: postscript (color), versatec (black-and-white or color), gremlin, and pixels (a generalized pixel-file that can be massaged in many ways). To plot part of your design in PostScript, place the box around the part you'd like to plot and type

:plot postscript

This will generate a plot of the area of the box. Everything visible underneath the box will appear in more-or-less the same way in the plot. *Width* specifies how wide the plot will be, in inches. Magic will scale the plot so that the area of the box comes out this wide. The default for *width* is the width of the plotter (if *width* is larger than the plotter width, it's reduced to the plotter width). If *layers* is given, it specifies exactly what information is to be plotted. Only those layers will appear in the plot. The special "layer" **labels** will enable label plotting.

The second form is for driving printers like color Versatecs. It is enabled by setting the *color* plot parameter to *true*. A table of stipples for the primary colors (black, cyan, magenta abd yellow) is given in the technology file. When the *plot* command is given, four rasters (one for each of the colors) are generated, separated with the proper control sequences for the printer. Otherwise, operation is exactly as for the black-and-white case.

The third form of plotting is for generating Gremlin-format files, which can then be edited with the Gremlin drawing system or included in documents processed by Grn and Ditroff. The command to get Gremlin files is

:plot gremlin *file* [*layers*]

It will generate a Gremlin-format file in *file* that describes everything underneath the box. If *layers* is specified, it indicates which layers are to appear in the file; otherwise everything visible on the screen is output. The Gremlin file is output without any particular scale; use the **width** or

height commands in Grn to scale the plot when it's printed. You should use the **mg** stipples when printing Magic Gremlin plots; these will produce the same stipple patterns as **:plot versatec**.

Finally, the "pixels" style of plotting generates a file of pixel values for the region to be plotted. This can be useful for input to other image tools, or for generation of slides and viewgraphs for presentations. The file consists of a sequence of bytes, three for each pixel, written from left to right and top to bottom. Each three bytes represent the red, green and blue values used to display the pixel. Thus, if the upper-left-most pixel were to be red, the first three bytes of the file would have values of 255, 0 and 0.

The resolution of the generated file is normally 512, but can be controlled by setting the plot parameter *pixWidth*. It must be a multiple of 8; Magic will round up if an inappropriate value is entered. The height of the file is determined by the shape of the box. In any case, the actual resolution of the file is appended to the file name. For example, plotting a square region, 2048 pixels across, will result in a file named something like "magicPlot1234a-2048-2048".

There are several plotting parameters used internally to Magic, such as the width of the Versatec printer and the number of dots per inch on the Versatec printer. You can modify most of these to work with different printers. For details, read about the various **:plot** command options in the *man* page.

10 Utility Commands

There are several additional commands that you will probably find useful once you start working on real cells. The command

:grid [*spacing*]
:grid *xSpacing ySpacing*
:grid *xSpacing ySpacing xOrigin yOrigin*
:grid off

will display a grid over your layout. Initially, the grid has a one-unit spacing. Typing **:grid** with no arguments will toggle the grid on and off. If a single numerical argument is given, the grid will be turned on, and the grid lines will be *spacing* units apart. The macro **g** provides a short form for **:grid** and **G** is short for **:grid 2**. If you provide two arguments to **:grid**, they are the x- and y-spacings, which may be different. If you provide four arguments, the last two specify a reference point through which horizontal and vertical grid lines pass; the default is to use (0,0) as the grid origin. The command **:grid off** always turns the grid off, regardless of whether or not is was previously on. When the grid is on, a small black box is displayed to mark the (0,0) coordinate of the cell you're editing.

If you want to create a cell that doesn't fit on the screen, you'll need to know how to change the screen view. This can be done with three commands:

:zoom *factor*
:findbox [**zoom**]
:view

If *factor* is given to the zoom command, it is a zoom-out factor. For example, the command **:zoom 2** will change the view so that there are twice as many units across the screen as there used to be (**Z** is a macro for this). The new view will have the same center as the old one. The command **:zoom .5** will increase the magnification so that only half as much of the circuit is visible.

The **:findbox** command is used to change the view according to the box. The command alone just moves the view (without changing the scale factor) so that the box is in the center of the screen. If the **zoom** argument is given then the magnification is changed too, so that the area of the box nearly fills the screen. **z** is a macro for **:findbox zoom** and **B** is a macro for **:findbox**.

The command **:view** resets the view so that the entire cell is visible in the window. It comes in handy if you get lost in a big layout. The macro **v** is equivalent to **:view**.

The command **:box** prints out the size and location of the box in case you'd like to measure something in your layout. The macro **b** is predefined to **:box**. The **:box** command can also be used to set the box to a particular location, height, or width. See the man page for details.

The command

:what

will print out information about what's selected. This may be helpful if you're not sure what layer a particular piece of material is, or what layer a particular label is attached to.

If you forget what a macro means, you can invoke the command

:macro [*char*]

This command will print out the long command that's associated with the macro *char*. If you omit *char*, Magic will print out all of the macro associations. The command

:macro *char command*

We set up *char* to be a macro for *command*, replacing the old *char* macro if there was one. If *command* contains any spaces then it must be enclosed in double-quotes. To see how this works, type the command **:macro** `1 "echo You just typed the 1 key."`, then type the 1 key.

One of the macros, "**.**", has special meaning in Magic. This macro is always defined by the system to be the last long command you typed. Whenever you'd like to repeat a long command, all you have to do is use the dot macro.

11 What the Layers Mean

The paint layers available in Magic are different from those that you may be used to in Caesar and other systems because they don't correspond exactly to the masks used in fabrication. We call them *abstract layers* because they correspond to constructs such as wires and contacts, rather than mask layers. We also call them *logs* because they look like sticks except that the geometry is drawn fully fleshed instead of as lines. In Magic there is one paint layer for each kind of conducting material (polysilicon, ndiffusion, metal1, etc.), plus one additional paint layer for each kind of transistor (ntransistor, ptransistor, etc.), and, finally, one further paint layer for each kind of contact (pcontact, ndcontact, m2contact, etc.) Each layer has one or more names that are used to refer to that layer in commands. To find out the layers available in the current technology, type the command

:layers

In addition to the mask layers, there are a few pseudo-layers that are valid in all technologies; these are listed in Table 1. Each Magic technology also has a technology manual describing the features of that technology, such as design rules, routing layers, CIF styles, etc. If you haven't seen any of the technology manuals yet, this is a good time to take a look at the one for your process.

errors (design-rule violations)
labels
subcells
(all mask layers)
$ (all mask layers visible under cursor)

Table 1: Pseudo-layers available in all technologies.

If you're used to designing with mask layers (e.g. you've been reading the Mead-Conway book), Magic's log style will take some getting used to. One of the reasons for logs is to save you work. In Magic you don't draw implants, wells, buried windows, or contact via holes. Instead, you draw the primary conducting layers and paint some of their overlaps with special types such as n-transistor or polysilicon contact. For transistors, you draw only the actual area of the transistor channel. Magic will generate the polysilicon and diffusion, plus any necessary implants, when it creates a CIF file. For contacts, you paint the contact layer in the area of overlap between the conducting layers. Magic will generate each of the constituent mask layers plus vias and buried windows when it writes the CIF file. Figure 1 shows a simple cell drawn with both mask layers (as in Caesar) and with logs (as in Magic). If you're curious about what the masks will look like for a particular layout, you can use the **:cif see** command to view the mask information.

An advantage of the logs used in Magic is that they simplify the design rules. Most of the formation rules (e.g. contact structure) go away, since Magic automatically generates correctly-formed structures when it writes CIF. All that are left are minimum size and spacing rules, and Magic's abstract layers result in fewer of these than there would be otherwise. This helps to make Magic's built-in design rule checker very fast (see "Magic Tutorial #6: Design Rule Checking"), and is one of the reasons plowing is possible.

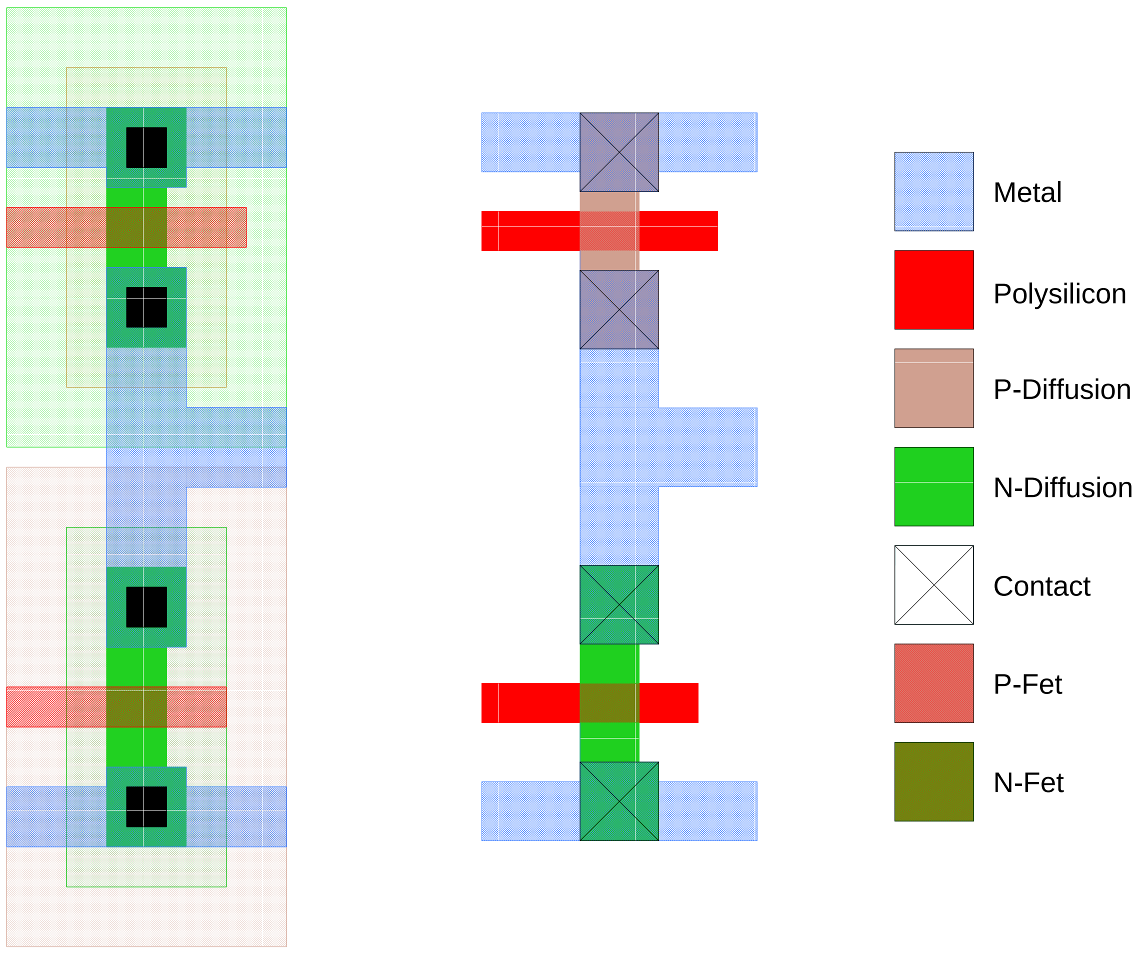


Figure 1: An example of how the logs are used. The figure on the left shows actual mask layers for an CMOS inverter cell, and the figure on the right shows the layers used to represent the cell in Magic.

Magic Tutorial #3: Advanced Painting (Wiring and Plowing)

John Ousterhout
Walter Scott

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection

Commands introduced in this tutorial:

:array, :corner, :fill, :flush, :plow, :straighten, :tool, :wire

Macros introduced in this tutorial:

<space>

1 Introduction

Tutorial #2 showed you the basic facilities for placing paint and labels, selecting, and manipulating the things that are selected. This tutorial describes two additional facilities for manipulating paint: wiring and plowing. These commands aren't absolutely necessary, since you can achieve the same effect with the simpler commands of Tutorial #2; however, wiring and plowing allow you to perform certain kinds of manipulations much more quickly than you could otherwise. Wiring is described in Section 2; it allows you to place wires by pointing at the ends of legs rather than by positioning the box, and also provides for convenient contact placement. Plowing is the subject of Section 3. It allows you to re-arrange pieces of your circuit without having to worry about design-rule violations being created: plowing automatically moves things out of the way to avoid trouble.

2 Wiring

The box-and-painting paradigm described in Tutorial #2 is sufficient to create any possible layout, but it's relatively inefficient since three keystrokes are required to paint each new area: two button clicks to position the box and one more to paint the material. This section describes a different painting mechanism based on *wires*. At any given time, there is a current wiring material and wire thickness. With the wiring interface you can create a new area of material with a single button click: this paints a straight-line segment of the current material and width between the end of the previous wire segment and the cursor location. Each additional button click adds an additional segment. The wiring interface also makes it easy for you to place contacts.

3 Tools

Before learning about wiring, you'll need to learn about tools. Until now, when you've pressed mouse buttons in layout windows the buttons have caused the box to change or material to be painted. The truth is that buttons can mean different things at different times. The meaning of the mouse buttons depends on the *current tool*. Each tool is identified by a particular cursor shape and a particular interpretation of the mouse buttons. Initially, the current tool is the box tool; when the box tool is active the cursor has the shape of a crosshair. To get information about the current tool, you can type the long command

:tool info

This command prints out the name of the current tool and the meaning of the buttons. Run Magic on the cell **tut3a** and type **:tool info**.

The **:tool** command can also be used to switch tools. Try this out by typing the command

:tool

Magic will print out a message telling you that you're using the wiring tool, and the cursor will change to an arrow shape. Use the **:tool info** command to see what the buttons mean now. You'll be using the wiring tool for most of the rest of this section. The macro " " (space) corresponds to **:tool**. Try typing the space key a few times: Magic will cycle circularly through all of the available tools. There are three tools in Magic right now: the box tool, which you already know about, the wiring tool, which you'll learn about in this tutorial, and the netlist tool, which has a square cursor shape and is used for netlist editing. "Tutorial #7: Netlists and Routing" will show you how to use the netlist tool.

The current tool affects only the meanings of the mouse buttons. It does not change the meanings of the long commands or macros. This means, for example, that you can still use all the selection commands while the wiring tool is active. Switch tools to the wiring tool, point at some paint in **tut3a**, and type the **s** macro. A chunk gets selected just as it does with the box tool.

4 Basic Wiring

There are three basic wiring commands: selecting the wiring material, adding a leg, and adding a contact. This section describes the first two commands. At this point you should be editing the cell **tut3a** with the wiring tool active. The first step in wiring is to pick the material and width to use for wires. This can be done in two ways. The easiest way is to find a piece of material of the right type and width, point to it with the cursor, and click the left mouse button. Try this in **tut3a** by pointing to the label **1** and left-clicking. Magic prints out the material and width that it chose, selects a square of that material and width around the cursor, and places the box around the square. Try pointing to various places in **tut3a** and left-clicking.

Once you've selected the wiring material, the right button paints legs of a wire. Left-click on label **1** to select the red material, then move the cursor over label **2** and right-click. This will paint a red wire between **1** and **2**. The new wire leg is selected so that you can modify it with selection commands, and the box is placed over the tip of the leg to show you the starting point for the next wire leg. Add more legs to the wire by right-clicking at **3** and then **4**. Use the mouse buttons to paint another wire in blue from **5** to **6** to **7**.

Each leg of a wire must be either horizontal or vertical. If you move the cursor diagonally, Magic will still paint a horizontal or vertical line (whichever results in the longest new wire leg). To see how this works, left-click on **8** in **tut3a**, then right-click on **9**. You'll get a horizontal leg. Now undo the new leg and right-click on **10**. This time you'll get a vertical leg. You can force Magic to paint the next leg in a particular direction with the commands

```
:wire horizontal
:wire vertical
```

Try out this feature by left-clicking on **8** in **tut3a**, moving the cursor over **10**, and typing **:wire ho** (abbreviations work for **:wire** command options just as they do elsewhere in Magic). This command will generate a short horizontal leg instead of a longer vertical one.

5 Contacts

When the wiring tool is active, the middle mouse button places contacts. Undo all of your changes to **tut3a** by typing the command **:flush** and answering **yes** to the question Magic asks. This throws away all of the changes made to the cell and re-loads it from disk. Draw a red wire leg from **1** to **2**. Now move the cursor over the blue area and click the middle mouse button. This has several effects. It places a contact at the end of the current wire leg, selects the contact, and moves the box over the selection. In addition, it changes the wiring material and thickness to match the material you middle-clicked. Move the cursor over **3** and right-click to paint a blue leg, then make a contact to purple by middle-clicking over the purple material. Continue by drawing a purple leg to **4**.

Once you've drawn the purple leg to **4**, move the cursor over red material and middle-click. This time, Magic prints an error message and treats the click just like a left-click. Magic only knows how to make contacts between certain combinations of layers, which are specified in the technology file (see "Magic Maintainer's Manual #2: The Technology File"). For this technology, Magic doesn't know how to make contacts directly between purple and red.

6 Wiring and the Box

In the examples so far, each new wire leg appeared to be drawn from the end of the previous leg to the cursor position. In fact, however, the new material was drawn from the *box* to the cursor position. Magic automatically repositions the box on each button click to help set things up for the next leg. Using the box as the starting point for wire legs makes it easy to start wires in places that don't already have material of the right type and width. Suppose that you want to start a new wire in the middle of an empty area. You can't left-click to get the wire started there. Instead, you can left-click some other place where there's the right material for the wire, type the space bar twice to get back the box tool, move the box where you'd like the wire to start, hit the space bar once more to get back the wiring tool, and then right-click to paint the wire. Try this out on **tut3a**.

When you first start wiring, you may not be able to find the right kind of material anywhere on the screen. When this happens, you can select the wiring material and width with the command

:wire type *layer width*

Then move the box where you'd like the wire to start, switch to the wiring tool, and right-click to add legs.

7 Wiring and the Selection

Each time you paint a new wire leg or contact using the wiring commands, Magic selects the new material just as if you had placed the cursor over it and typed **s**. This makes it easy for you to adjust its position if you didn't get it right initially. The **:stretch** command is particularly useful for this. In **tut3a**, paint a wire leg in blue from **5** to **6** (use **:flush** to reset the cell if you've made a lot of changes). Now type **R** two or three times to stretch the leg over to the right. Middle-click over purple material, then use **W** to stretch the contact downward.

It's often hard to position the cursor so that a wire leg comes out right the first time, but it's usually easy to tell whether the leg is right once it's painted. If it's wrong, then you can use the stretching commands to shift it over one unit at a time until it's correct.

8 Bundles of Wires

Magic provides two additional commands that are useful for running *bundles* of parallel wires. The commands are:

fill *direction* [*layers*]
corner *direction1 direction2* [*layers*]

To see how they work, load the cell **tut3b**. The **:fill** comand extends a whole bunch of paint in a given direction. It finds all paint touching one side of the box and extends that paint to the opposite side of the box. For example, **:fill left** will look underneath the right edge of the box for paint, and will extend that paint to the left side of the box. The effect is just as if all the colors visible underneath that edge of the box constituted a paint brush; Magic sweeps the brush across the box in the given direction. Place the box over the label "Fill here" in **tut3b** and type **:fill left**.

The **:corner** command is similar to **:fill** except that it generates L-shaped wires that follow two sides of the box, travelling first in *direction1* and then in *direction2*. Place the box over the label "Corner here" in **tut3b** and type **:corner right up**.

In both **:fill** and **:corner**, if *layers* isn't specified then all layers are filled. If *layers* is given then only those layers are painted. Experiment on **tut3b** with the **:fill** and **:corner** commands.

When you're painting bundles of wires, it would be nice if there were a convenient way to place contacts across the whole bundle in order to switch to a different layer. There's no single command to do this, but you can place one contact by hand and then use the **:array** command to replicate a single contact across the whole bundle. Load the cell **tut3c**. This contains a bundle of wires with a single contact already painted by hand on the bottom wire. Type **s** with the cursor over the contact, and type **S** with the cursor over the stub of purple wiring material next to it. Now place the box over the label "Array" and type the command **:array 1 10**. This will copy the selected contact across the whole bundle.

The syntax of the **:array** command is

:array *xsize ysize*

This command makes the selection into an array of identical elements. *Xsize* specifies how many total instances there should be in the x-direction when the command is finished and *ysize* specifies how many total instances there should be in the y-direction. In the **tut3c** example, **xsize** was one, so no additional copies were created in that direction; **ysize** was 10, so 9 additional copies were created. The box is used to determine how far apart the elements should be: the width of the box determines the x-spacing and the height determines the y-spacing. The new material always appears above and to the right of the original copy.

In **tut3c**, use **:corner** to extend the purple wires and turn them up. Then paint a contact back to blue on the leftmost wire, add a stub of blue paint above it, and use **:array** to copy them across the top of the bundle. Finally, use **:fill** again to extend the blue bundle farther up.

9 Plowing

Magic contains a facility called *plowing* that you can use to stretch and compact cells. The basic plowing command has the syntax

:plow *direction* [*layers*]

where *direction* is a Manhattan direction like **left** and *layers* is an optional, comma-separated list of mask layers. The plow command treats one side of the box as if it were a plow, and shoves the plow over to the other side of the box. For example, **:plow up** treats the bottom side of the box as a plow, and moves the plow to the top of the box.

As the plow moves, every edge in its path is pushed ahead of it (if *layers* is specified, then only edges on those layers are moved). Each edge that is pushed by the plow pushes other edges ahead of it in a way that preserves design rules, connectivity, and transistor and contact sizes. This means that material ahead of the plow gets compacted down to the minimum spacing permitted by the design rules, and material that crossed the plow's original position gets stretched behind the plow.

You can compact a cell by placing a large plow off to one side of the cell and plowing across the whole cell. You can open up space in the middle of a cell by dragging a small plow across the area where you want more space.

To try out plowing, edit the cell **tut3d**, place the box over the rectangle that's labelled “Plow here”, and try plowing in various directions. Also, try plowing only certain layers. For example, with the box over the “Plow here” label, try

:plow right metal2

Nothing happens. This is because there are no metal2 *edges* in the path of the plow. If instead you had typed

:plow right metal1

only the metal would have been plowed to the right.

In addition to plowing with the box, you can plow the selection. The command to do this has the following syntax:

:plow selection [*direction* [*distance*]]

This is very similar to the **:stretch** command: it picks up the selection and the box and moves both so that the lower-left corner of the box is at the cursor location. Unlike the **:stretch** command, though, **:plow selection** insures that design rule correctness and connectivity are preserved.

Load the cell **tut3e** and use **a** to select the area underneath the label that says “select me”. Then point with the cursor to the point labelled “point here” and type **:plow selection**. Practice selecting things and plowing them. Like the **:stretch** command, there is also a longer form of **:plow selection**. For example, **:plow selection down 5** will plow the selection and the box down 10 units.

Selecting a cell and plowing it is a good way to move the cell. Load **tut3f** and select the cell **tut3e**. Point to the label “point here” and plow the selection with **:plow selection**. Notice that all connections to the cell have remained attached. The cell you select must be in the edit cell, however.

The plowing operation is implemented in a way that tries to keep your design as compact as possible. To do this, it inserts jogs in wires around the plow. In many cases, though, the additional jogs are more trouble than they're worth. To reduce the number of jogs inserted by plowing, type the command

:plow nojogs

From now on, Magic will insert as few jogs as possible when plowing, even if this means moving more material. You can re-enable jog insertion with the command

:plow jogs

Load the cell **tut3d** again and try plowing it both with and without jog insertion.

There is another way to reduce the number of jogs introduced by plowing. Instead of avoiding jogs in the first place, plowing can introduce them freely but clean them up as much as possible afterward. This results in more dense layouts, but possibly more jogs than if you had enabled **:plow nojogs**. To take advantage of this second method for jog reduction, re-enable jog insertion (**:plow jogs**) and enable jog cleanup with the command

:plow straighten

From now on, Magic will attempt to straighten out jogs after each plow operation. To disable straightening, use the command

:plow nostraighten

It might seem pointless to disable jog introduction with **:plow nojogs** at the same time straightening is enabled with **:plow straighten**. While it is true that **:plow nojogs** won't introduce any new jogs for **:plow straighten** to clean up, plowing will straighten out any existing jogs after each operation.

In fact, there is a separate command that is sometimes useful for cleaning up layouts with many jogs, namely the command

:straighten *direction*

where *direction* is a Manhattan direction, e.g., **up**, **down**, **right**, or **left**. This command will start from one side of the box and pull jogs toward that side to straighten them. Load the cell **tut3g**, place the box over the label "put box here", and type **:straighten left**. Undo the last command and type **:straighten right** instead. Play around with the **:straighten** command.

There is one more feature of plowing that is sometimes useful. If you are working on a large cell and want to make sure that plowing never affects any geometry outside of a certain area, you can place a *boundary* around the area you want to affect with the command

:plow boundary

The box is used to specify the area you want to affect. After this command, subsequent plows will only affect the area inside this boundary.

Load the cell **tut3h** place the box over the label "put boundary here", and type **:plow boundary**. Now move the box away. You will see the boundary highlighted with dotted lines. Now place the box over the area labelled "put plow here" and plow up. This plow would cause geometry outside of the boundary to be affected, so Magic reduces the plow distance enough to prevent this and warns you of this fact. Now undo the last plow and remove the boundary with

:plow noboundary

Put the box over the "put plow here" label and plow up again. This time there was no boundary to stop the plow, so everything was moved as far as the height of the box. Experiment with placing the boundary around an area of this cell and plowing.

Magic Tutorial #4: Cell Hierarchies

John Ousterhout

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection

Commands introduced in this tutorial:

:array, :edit, :expand, :flush, :getcell, :identify, :load, :path, :see, :unexpand

Macros introduced in this tutorial:

x, X, ˆX

1 Introduction

In Magic, a layout is a hierarchical collection of cells. Each cell contains three things: paint, labels, and subcells. Tutorial #2 showed you how to create and edit paint and labels. This tutorial describes Magic's facilities for building up cell hierarchies. Strictly speaking, hierarchical structure isn't necessary: any design that can be represented hierarchically can also be represented "flat" (with all the paint and labels in a single cell). However, many things are greatly improved if you use a hierarchical structure, including the efficiency of the design tools, the speed with which you can enter the design, and the ease with which you can modify it later.

2 Selecting and Viewing Hierarchical Designs

“Hierarchical structure” means that each cell can contain other cells as components. To look at an example of a hierarchical layout, enter Magic with the shell command **magic tut4a**. The cell **tut4a** contains four subcells plus some blue paint. Two of the subcells are instances of cell **tut4x** and two are instances of **tut4y**. Initially, each subcell is displayed in *unexpanded* form. This means that no details of the subcell are displayed; all you see is the cell’s bounding box, plus two names inside the bounding box. The top name is the name of the subcell (the name you would type when invoking Magic to edit the cell). The cell’s contents are stored in a file with this name plus a **.mag** extension. The bottom name inside each bounding box is called an *instance identifier*, and is used to distinguish different instances of the same subcell. Instance id’s are used for routing and circuit extraction, and are discussed in Section 6.

Subcells can be manipulated using the same selection mechanism that you learned in Tutorial #2. To select a subcell, place the cursor over the subcell and type **f** (“find cell”), which is a macro for **:select cell**. You can also select a cell by typing **s** when the cursor is over a location where there’s no paint; **f** is probably more convenient, particularly for cells that are completely covered with paint. When you select a cell the box will be set to the cell’s bounding box, the cell’s name will be highlighted, and a message will be printed on the text display. All the selection operations (**:move**, **:copy**, **:delete**, etc.) apply to subcells. Try selecting and moving the top subcell in **tut4a**. You can also select subcells using area selection (the **a** and **A** macros): any unexpanded subcells that intersect the area of the box will be selected.

To see what’s inside a cell instance, you must *expand* it. Select one of the instances of **tut4y**, then type the command

:expand toggle

or invoke the macro **ˆX** which is equivalent. This causes the internals of that instance of **tut4y** to be displayed. If you type **ˆX** again, the instance is unexpanded so you only see a bounding box again. The **:expand toggle** command expands all of the selected cells that are unexpanded, and unexpands all those that are expanded. Type **ˆX** a third time so that **tut4y** is expanded.

As you can see now, **tut4y** contains an array of **tut4x** cells plus some additional paint. In Magic, an array is a special kind of instance containing multiple copies of the same subcell spaced at fixed intervals. Arrays can be one-dimensional or two-dimensional. The whole array is always treated as a single instance: any command that operates on one element of the array also operates on all the other elements simultaneously. The instance identifiers for the elements of the array are the same except for an index. Now select one of the elements of the array and expand it. Notice that the entire array is expanded at the same time.

When you have expanded the array, you’ll see that the paint in the top-level cell **tut4a** is displayed more brightly than the paint in the **tut4x** instances. **Tut4a** is called the *edit cell*, because its contents are currently editable. The paint in the edit cell is normally displayed more brightly than other paint to make it clear that you can change it. As long as **tut4a** is the edit cell, you cannot modify the paint in **tut4x**. Try erasing paint from the area of one of the **tut4x** instances: nothing will be changed. Section 4 tells how to switch the edit cell.

Place the cursor over one of the **tut4x** array elements again. At this point, the cursor is actually over three different cells: **tut4x** (an element of an array instance within **tut4y**), **tut4y** (an instance within **tut4a**), and **tut4**. Even the topmost cell in the hierarchy is treated as an instance by Magic. When you press the **s** key to select a cell, Magic initially chooses the smallest instance visible underneath the cursor, **tut4x** in this case. However, if you invoke the **s** macro again (or type **:select**) without moving the cursor, Magic will step through all of the instances under the cursor in order. Try this out. The same is true of the **f** macro and **:select cell**.

When there are many different expanded cells on the screen, you can use the selection commands to select paint from any of them. You can select anything that's visible, regardless of which cell it's in. However, as mentioned above, you can only modify paint in the edit cell. If you use **:move** or **:upsidedown** or similar commands when you've selected information outside the edit cell, the information outside the edit cell is removed from the selection before performing the operation.

There are two additional commands you can use for expanding and unexpanding cells:

> **:expand**
> **:unexpand**

Both of these commands operate on the area underneath the box. The **:expand** command will recursively expand every cell that intersects the box until there are no unexpanded cells left under the box. The **:unexpand** command will unexpand every cell whose area intersects the box but doesn't completely contain it. The macro **x** is equivalent to **:expand**, and **X** is equivalent to **:unexpand**. Try out the various expansion and unexpansion facilities on **tut4a**.

3 Manipulating Subcells

There are a few other commands, in addition to the selection commands already described, that you'll need in order to manipulate subcells. The command

> **:getcell** *name*

will find the file *name***.mag** on disk, read the cell it contains, and create an instance of that cell with its lower-left corner aligned with the lower-left corner of the box. Use the **getcell** command to get an instance of the cell **tut4z**. After the **getcell** command, the new instance is selected so you can move it or copy it or delete it. The **getcell** command recognizes additional arguments that permit the cell to be positioned using labels and/or explicit coordinates. See the *man* page for details.

To turn a normal instance into an array, select the instance and then invoke the **:array** command. It has two forms:

> **:array** *xsize ysize*
> **:array** *xlo xhi ylo yhi*

In the first form, *xsize* indicates how many elements the array should have in the x-direction, and *ysize* indicates how many elements it should have in the y-direction. The spacing between elements is controlled by the box's width (for the x-direction) and height (for the y-direction). By changing the box size, you can space elements so that they overlap, abut, or have gaps between them. The elements are given indices from 0 to *xsize*-1 in the x-direction and from 0 to *ysize*-1 in the y-direction. The second form of the command is identical to the first except that the elements are given indices from *xlo* to *xhi* in the x-direction and from *ylo* to *yhi* in the y-direction. Try making a 4x4 array out of the **tut4z** cell with gaps between the cells.

You can also invoke the **:array** command on an existing array to change the number of elements or spacing. Use a size of 1 for *xsize* or *ysize* in order to get a one-dimensional array. If there are several cells selected, the **:array** command will make each of them into an array of the same size and spacing. It also works on paint and labels: if paint and labels are selected when you invoke **:array**, they will be copied many times over to create the array. Try using the array command to replicate a small strip of paint.

4 Switching the Edit Cell

At any given time, you are editing the definition of a single cell. This definition is called the *edit cell*. You can modify paint and labels in the edit cell, and you can re-arrange its subcells. You may not re-arrange or delete the subcells of any cells other than the edit cell, nor may you modify the paint or labels of any cells except the edit cell. You may, however, copy information from other cells into the edit cell, using the selection commands. To help clarify what is and isn't modifiable, Magic displays the paint of the edit cell in brighter colors than other paint.

When you rearrange subcells of the edit cell, you aren't changing the subcells themselves. All you can do is change the way they are used in the edit cell (location, orientation, etc.). When you delete a subcell, nothing happens to the file containing the subcell; the command merely deletes the instance from the edit cell.

Besides the edit cell, there is one other special cell in Magic. It's called the *root cell* and is the topmost cell in the hierarchy, the one you named when you ran Magic (**tut4a** in this case). As you will see in Tutorial #5, there can actually be several root cells at any given time, one in each window. For now, there is only a single window on the screen, and thus only a single root cell. The window caption at the top of the color display contains the name of the window's root cell and also the name of the edit cell.

Up until now, the root cell and the edit cell have been the same. However, this need not always be the case. You can switch the edit cell to any cell in the hierarchy by selecting an instance of the definition you'd like to edit, and then typing the command

:edit

Use this command to switch the edit cell to one of the **tut4x** instances in **tut4a**. Its paint brightens, while the paint in **tut4a** becomes dim. If you want to edit an element of an array, select the array, place the cursor over the element you'd like to edit, then type **:edit**. The particular element underneath the cursor becomes the edit cell.

When you edit a cell, you are editing the master definition of that cell. This means that if the cell is used in several places in your design, the edits will be reflected in all those places. Try painting and erasing in the **tut4x** cell that you just made the edit cell: the modifications will appear in all of its instances.

There is a second way to change the edit cell. This is the command

:load *name*

The **:load** command loads a new hierarchy into the window underneath the cursor. *Name* is the name of the root cell in the hierarchy. If no *name* is given, a new unnamed cell is loaded and you start editing from scratch. The **:load** command only changes the edit cell if there is not already an edit cell in another window.

5 Subcell Usage Conventions

Overlaps between cells are occasionally useful to share busses and control lines running along the edges. However, overlaps cause the analysis tools to work much harder than they would if there were no overlaps: wherever cells overlap, the tools have to combine the information from the two separate cells. Thus, you shouldn't use overlaps any more than absolutely necessary. For example, suppose you want to create a one-dimensional array of cells that alternates between two cell types, A and B: "ABABABABABAB". One way to do this is first to make an array of A instances with large gaps between them ("A A A A A A"), then make an array of B instances with large gaps between them ("B B B B B B"), and finally place one array on top of the other so that the B's nestle in between the A's. The problem with this approach is that the two arrays overlap almost completely, so Magic will have to go to a lot of extra work to handle the overlaps (in this case, there isn't much overlap of actual paint, but Magic won't know this and will spend a lot of time worrying about it). A better solution is to create a new cell that contains one instance of A and one instance of B, side by side. Then make an array of the new cell. This approach makes it clear to Magic that there isn't any real overlap between the A's and B's.

If you do create overlaps, you should use the overlaps only to connect the two cells together, and not to change their structure. This means that the overlap should not cause transistors to appear, disappear, or change size. The result of overlapping the two subcells should be the same electrically as if you placed the two cells apart and then ran wires to hook parts of one cell to parts of the other. The convention is necessary in order to be able to do hierarchical circuit extraction easily (it makes it possible for each subcell to be circuit-extracted independently).

Three kinds of overlaps are flagged as errors by the design-rule checker. First, you may not overlap polysilicon in one subcell with diffusion in another cell in order to create transistors. Second, you may not overlap transistors or contacts in one cell with different kinds of transistors or contacts in another cell (there are a few exceptions to this rule in some technologies). Third, if contacts from different cells overlap, they must be the same type of contact and must coincide exactly: you may not have partial overlaps. This rule is necessary in order to guarantee that Magic can generate CIF for fabrication.

You will make life a lot easier on yourself (and on Magic) if you spend a bit of time to choose a clean hierarchical structure. In general, the less cell overlap the better. If you use extensive overlaps you'll find that the tools run very slowly and that it's hard to make modifications to the circuit.

6 Instance Identifiers

Instance identifiers are used to distinguish the different subcells within a single parent. The cell definition names cannot be used for this purpose because there could be many instances of a single definition. Magic will create default instance id's for you when you create new instances with the **:get** or **:copy** commands. The default id for an instance will be the name of the definition with a unique integer added on. You can change an id by selecting an instance (which must be a child of the edit cell) and invoking the command

:identify *newid*

where *newid* is the identifier you would like the instance to have. *Newid* must not already be used as an instance identifier of any subcell within the edit cell.

Any node or instance can be described uniquely by listing a path of instance identifiers, starting from the root cell. The standard form of such names is similar to Unix file names. For example, if **id1** is the name of an instance within the root cell, **id2** is an instance within **id1**, and **node** is a node name within **id2**, then **id1/id2/node** can be used unambiguously to refer to the node. When you select a cell, Magic prints out the complete path name of the instance.

Arrays are treated specially. When you use **:identify** to give an array an instance identifier, each element of the array is given the instance identifier you specified, followed by one or two array subscripts enclosed in square brackets, e.g, **id3[2]** or **id4[3][7]**. When the array is one-dimensional, there is a single subscript; when it is two-dimensional, the first subscript is for the y-dimension and the second for the x-dimension.

7 Writing and Flushing Cells

When you make changes to your circuit in Magic, there is no immediate effect on the disk files that hold the cells. You must explicitly save each cell that has changed, using either the **:save** command or the **:writeall** command. Magic keeps track of the cells that have changed since the last time they were saved on disk. If you try to leave Magic without saving all the cells that have changed, the system will warn you and give you a chance to return to Magic to save them. Magic never flushes cells behind your back, and never throws away definitions that it has read in. Thus, if you edit a cell and then use **:load** to edit another cell, the first cell is still saved in Magic even though it doesn't appear anywhere on the screen. If you then invoke **:load** a second time to go back to the first cell, you'll get the edited copy.

If you decide that you'd really like to discard the edits you've made to a cell and recover the old version, there are two ways you can do it. The first way is using the **flush** option in **:writeall**. The second way is to use the command

:flush [*cellname*]

If no *cellname* is given, then the edit cell is flushed. Otherwise, the cell named *cellname* is flushed. The **:flush** command will expunge Magic's internal copy of the cell and replace it with the disk copy.

When you are editing large chips, Magic may claim that cells have changed even though you haven't modified them. Whenever you modify a cell, Magic makes changes in the parents of the cell, and their parents, and so on up to the root of the hierarchy. These changes record new design-rule violations, as well as timestamp and bounding box information used by Magic to keep track of design changes and enable fast cell read-in. Thus, whenever you change one cell you'll generally need to write out new copies of its parents and grandparents. If you don't write out the parents, or if you edit a child "out of context" (by itself, without the parents loaded), then you'll incur extra overhead the next time you try to edit the parents. "Timestamp mismatch" warnings are printed when you've edited cells out of context and then later go back and read in the cell as part of its parent. These aren't serious problems; they just mean that Magic is doing extra work to update information in the parent to reflect the child's new state.

8 Search Paths

When many people are working on a large design, the design will probably be more manageable if different pieces of it can be located in different directories of the file system. Magic provides a simple mechanism for managing designs spread over several directories. The system maintains a *search path* that tells which directories to search when trying to read in cells. By default, the search path is ".", which means that Magic looks only in the working directory. You can change the path using the command

:path [*searchpath*]

where *searchpath* is the new path that Magic should use. *Searchpath* consists of a list of directories separated by colons. For example, the path ".:˜ouster/x:a/b" means that if Magic is trying to read in a cell named "foo", it will first look for a file named "foo.mag" in the current directory. If it doesn't find the file there, it will look for a file named "˜ouster/x/foo.mag", and if that doesn't exist, then it will try "a/b/foo.mag" last. To find out what the current path is, type **:path** with no arguments. In addition to your path, this command will print out the system cell library path (where Magic looks for cells if it can't find them anywhere in your path), and the system search path (where Magic looks for files like colormaps and technology files if it can't find them in your current directory).

If you're working on a large design, you should use the search path mechanism to spread your layout over several directories. A typical large chip will contain a few hundred cells; if you try to place all of them in the same directory there will just be too many things to manage. For example, place the datapath in one directory, the control unit in another, the instruction buffer in a third, and so on. Try to keep the size of each directory down to a few dozen files. You can place the **:path** command in a **.magic** file in your home directory or the directory you normally run Magic from; this will save you from having to retype it each time you start up (see the Magic man page to find out about **.magic** files). If all you want to do is add another directory onto the end of the search path, you can use the **:addpath** [*directory*] command.

Because there is only a single search path that is used everywhere in Magic, you must be careful not to re-use the same cell name in different portions of the chip. A common problem with large designs is that different designers use the same name for different cells. This works fine as long as the designers are working separately, but when the two pieces of the design are put together using a search path, a single copy of the cell (the one that is found first in the search path) gets used everywhere.

There's another caveat in the use of search paths. Magic looks for system files in ˜cad, but sometimes it is helpful to put Magic's system files elsewhere. If the **CAD_HOME** shell environment variable is set, then Magic uses that as the location of ˜cad instead of the location in the password file. This overrides all uses of ˜cad within magic, including the ˜cad seen in the search paths printed out by **:path**.

9 Additional Commands

This section describes a few additional cell-related commands that you may find useful. One of them is the command

:select save *file*

This command takes the selection and writes it to disk as a new Magic cell in the file *file***.mag**. You can use this command to break up a big file into smaller ones, or to extract pieces from an existing cell.

The command

:dump *cellName* [*labelName*]

does the opposite of **select save**: it copies the contents of cell *cellName* into the edit cell, such that the lower-left corner of label *labelName* is at the lower-left corner of the box. The new material will also be selected. This command is similar in form to the **getcell** command except that it copies the contents of the cell instead of using the cell as a subcell. There are several forms of **dump**; see the *man* page for details.

The main purpose of **dump** is to allow you to create a library of cells representing commonly-used structures such as standard transistor shapes or special contact arrangements. You can then define macros that invoke the **dump** command to place the cells. The result is that a single keystroke is all you need to copy one of them into the edit cell.

As mentioned earlier, Magic normally displays the edit cell in brighter colors than non-edit cells. This helps to distinguish what is editable from what is not, but may make it hard for you to view non-edit paint since it appears paler. If you type the command

:see allSame

you'll turn off this feature: all paint everywhere will be displayed in the bright colors. The word **allSame** must be typed just that way, with one capital letter. If you'd like to restore the different display styles, type the command

:see no allSame

You can also use the **:see** command to selectively disable display of various mask layers in order to make the other ones easier to see. For details, read about **:see** in the Magic man page.

Magic Tutorial #5: Multiple Windows

Robert N. Mayo

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection

Commands introduced in this tutorial:

:center :closewindow, :openwindow, :over, :specialopen, :under, :windowpositions

Macros introduced in this tutorial:

o, O, “;”

1 Introduction

A window is a rectangular viewport. You can think of it as a magnifying glass that may be moved around on your chip. Magic initially displays a single window on the screen. This tutorial will show you how to create new windows and how to move old ones around. Multiple windows allow you to view several portions of a circuit at the same time, or even portions of different circuits.

Some operations are easier with multiple windows. For example, let’s say that you want to paint a very long line, say 3 units by 800 units. With a single window it is hard to align the box accurately since the magnification is not great enough. With multiple windows, one window can show the big picture while other windows show magnified views of the areas where the box needs to be aligned. The box can then be positioned accurately in these magnified windows.

2 Manipulating Windows

2.1 Opening and Closing Windows

Initially Magic displays one large window. The

:openwindow [*cellname*]

command opens another window and loads the given cell. To give this a try, start up Magic with the command **magic tut5a**. Then point anywhere in a Magic window and type the command **:openwindow tut5b** (make sure you're pointing to a Magic window). A new window will appear and it will contain the cell **tut5b**. If you don't give a *cellname* argument to **:openwindow**, it will open a new window on the cell containing the box, and will zoom in on the box. The macro **o** is predefined to **:openwindow**. Try this out by placing the box around an area of **tut5b** and then typing **o**. Another window will appear. You now have three windows, all of which display pieces of layout. There are other kinds of windows in Magic besides layout windows: you'll learn about them later. Magic doesn't care how many windows you have (within reason) nor how they overlap.

To get rid of a window, point to it and type

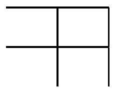
:closewindow

or use the macro **O**. Point to a portion of the original window and close it.

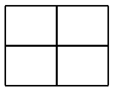
2.2 Resizing and Moving Windows

If you have been experimenting with Magic while reading this you will have noticed that windows opened by **:openwindow** are all the same size. If you'd prefer a different arrangement you can resize your windows or move them around on the screen. The techniques used for this are different, however, depending on what kind of display you're using. If you are using a workstation, then you are also running a window system such as X11 or SunView. In this case Magic's windows are moved and resized just like the other windows you have displayed, and you can skip the rest of this section.

For displays like the AED family, which don't have a built-in window package, Magic implements its own window manager. To re-arrange windows on the screen you can use techniques similar to those you learned for moving the box for painting operations. Point somewhere in the border area of a window, except for the lower left corner, and press and hold the right button. The cursor will change to a shape like this:



This indicates that you have hold of the upper right corner of the window. Point to a new location for this corner and release the button. The window will change shape so that the corner moves. Now point to the border area and press and hold the left button. The cursor will now look like:



This indicates that you have hold of the entire window by its lower left window. Move the cursor and release the button. The window will move so that its lower left corner is where you pointed.

The other button commands for positioning the box by any of its corners also work for windows. Just remember to point to the border of a window before pushing the buttons.

The middle button can be used to grow a window up to full-screen size. To try this, click the middle button over the caption of the window. The window will now fill the entire screen. Click in the caption again and the window will shrink back to its former size.

2.3 Shuffling Windows

By now you know how to open, close, and resize windows. This is sufficient for most purposes, but sometimes you want to look at a window that is covered up by another window. The **:underneath** and **:over** commands help with this.

The **:underneath** command moves the window that you are pointing at underneath all of the other windows. The **:over** command moves the window on top of the rest. Create a few windows that overlap and then use these commands to move them around. You'll see that overlapping windows behave just like sheets of paper: the ones on top obscure portions of the ones underneath.

2.4 Scrolling Windows

Some of the windows have thick bars on the left and bottom borders. These are called *scroll bars*, and the slugs within them are called *elevators*. The size and position of an elevator indicates how much of the layout (or whatever is in the window) is currently visible. If an elevator fills its scroll bar, then all of the layout is visible in that window. If an elevator fills only a portion of the scroll bar, then only that portion of the layout is visible. The position of the elevator indicates which part is visible—if it is near the bottom, you are viewing the bottom part of the layout; if it is near the top, you are viewing the top part of the layout. There are scroll bars for both the vertical direction (the left scroll bar) and the horizontal direction (the bottom scroll bar).

Besides indicating how much is visible, the scroll bars can be used to change the view of the window. Clicking the middle mouse button in a scroll bar moves the elevator to that position. For example, if you are viewing the lower half of a chip (elevator near the bottom) and you click the middle button near the top of the scroll bar, the elevator will move up to that position and you will be viewing the top part of your chip. The little squares with arrows in them at the ends of the scroll bars will scroll the view by one screenful when the middle button is clicked on them. They are useful when you want to move exactly one screenful. The **:scroll** command can also be used to scroll the view (though we don't think it's as easy to use as the scroll bars). See the man page for information on it.

If you only want to make a small adjustment in a window's view, you can use the command

:center

It will move the view in the window so that the point that used to be underneath the cursor is now in the middle of the window. The macro **,** is predefined to **:center**.

The bull's-eye in the lower left corner of a window is used to zoom the view in and out. Clicking the left mouse button zooms the view out by a factor of 2, and clicking the right mouse button zooms in by a factor of 2. Clicking the middle button here makes everything in the window visible and is equivalent to the **:view** command.

2.5 Saving Window Configurations

After setting up a bunch of windows you may want to save the configuration (for example, you may be partial to a set of 3 non-overlapping windows). To do this, type:

:windowpositions *filename*

A set of commands will be written to the file. This file can be used with the **:source** command to recreate the window configuration later. (However, this only works well if you stay on the same kind of display; if you create a file under X11 and then **:source** it under SunView, you might not get the same positions since the coordinate systems may vary.)

3 How Commands Work Inside of Windows

Each window has a caption at the top. Here is an example:

mychip EDITING shiftcell

This indicates that the window contains the root cell **mychip**, and that a subcell of it called **shiftcell** is being edited. You may remember from the Tutorial #4 that at any given time Magic is editing exactly one cell. If the edit cell is in another window then the caption on this window will read:

mychip [NOT BEING EDITED]

Let's do an example to see how commands are executed within windows. Close any layout windows that you may have on the screen and open two new windows, each containing the cell **tut5a**. (Use the **:closewindow** and **:openwindow tut5a** commands to do this.) Try moving the box around in one of the windows. Notice that the box also moves in the other window. Windows containing the same root cell are equivalent as far as the box is concerned: if it appears in one it will appear in all, and it can be manipulated from them interchangeably. If you change **tut5a** by painting or erasing portions of it you will see the changes in both windows. This is because both windows are looking at the same thing: the cell **tut5a**. Go ahead and try some painting and erasing until you feel comfortable with it. Try positioning one corner of the box in one window and another corner in another window. You'll find it doesn't matter which window you point to, all Magic knows is that you are pointing to **tut5a**.

These windows are independent in some respects, however. For example, you may scroll one window around without affecting the other window. Use the scrollbars to give this a try. You can also expand and unexpand cells independently in different windows.

We have seen how Magic behaves when both windows view a single cell. What happens when windows view different cells? To try this out load **tut5b** into one of the windows (point to a window and type **:load tut5b**). You will see the captions on the windows change—only one window contains the cell currently being edited. The box cannot be positioned by placing one corner in one window and another corner in the other window because that doesn't really make sense (try it). However, the selection commands work between windows: you can select information in one window and then copy it into another (this only works if the window you're copying into contains the edit cell; if not, you'll have to use the **:edit** command first).

The operation of many Magic commands is dependent upon which window you are pointing at. If you are used to using Magic with only one window you may, at first, forget to point to the window that you want the operation performed upon. For instance, if there are several windows on the screen you will have to point to one before executing a command like **:grid**—otherwise you may not affect the window that you intended!

4 Special Windows

In addition to providing multiple windows on different areas of a layout, Magic provides several special types of windows that display things other than layouts. For example, there are special window types to edit netlists and to adjust the colors displayed on the screen. One of the special window types is described in the section below; others are described in the other tutorials. The

:specialopen *type* [*args*]

command is used to create these sorts of windows. The *type* argument tells what sort of window you want, and *args* describe what you want loaded into that window. The **:openwindow** *cellname* command is really just short for the command **:specialopen layout** *cellname*.

Each different type of window (layout, color, etc.) has its own command set. If you type **:help** in different window types, you'll see that the commands are different. Some of the commands, such as those to manipulate windows, are valid in all windows, but for other commands you must make sure you're pointing to the right kind of window or the command may be misinterpreted. For example, the **:extract** command means one thing in a layout window and something totally different in a netlist window.

5 Color Editing

Special windows of type **color** are used to edit the red, green, and blue intensities of the colors displayed on the screen. To create a color editing window, invoke the command

:specialopen color [*number*]

Number is optional; if present, it gives the octal value of the color number whose intensities are to be edited. If *number* isn't given, 0 is used. Try opening a color window on color 0.

A color editing window contains 6 "color bars", 12 "color pumps" (one on each side of each bar), plus a large rectangle at the top of the window that displays a swatch of the color being edited (called the "current color" from now on). The color bars display the components of the current color in two different ways. The three bars on the left display the current color in terms of its red, green, and blue intensities (these intensities are the values actually sent to the display). The three bars on the right display the current color in terms of hue, saturation, and value. Hue selects a color of the spectrum. Saturation indicates how diluted the color is (high saturation corresponds to a pure color, low saturation corresponds to a color that is diluted with gray, and a saturation of 0 results in gray regardless of hue). Value indicates the overall brightness (a value of 0 corresponds to black, regardless of hue or saturation).

There are several ways to modify the current color. First, try pressing any mouse button while the cursor is over one of the color bars. The length of the bar, and the current color, will be modified to reflect the mouse position. The color map in the display is also changed, so the colors will change everywhere on the screen that the current color is displayed. Color 0, which you should currently be editing, is the background color. You can also modify the current color by pressing a button while the cursor is over one of the "color pumps" next to the bars. If you button a pump with "+" in it, the value of the bar next to it will be incremented slightly, and if you button the "-" pump, the bar will be decremented slightly. The left button causes a change of about 1% in the value of the bar, and the right button will pump the bar up or down by about 5%. Try adjusting the bars by buttoning the bars and the pumps.

If you press a button while the cursor is over the current color box at the top of the window, one of two things will happen. In either case, nothing happens until you release the button. Before releasing the button, move the cursor so it is over a different color somewhere on the screen. If you pressed the left button, then when the button is released the color underneath the cursor becomes the new current color, and all future editing operations will affect this color. Try using this feature to modify the color used for window borders. If you pressed the right button, then when the button is released the value of the current color is copied from whatever color is present underneath the cursor.

There are only a few commands you can type in color windows, aside from those that are valid in all windows. The command

:color [*number*]

will change the current color to *number*. If no *number* is given, this command will print out the current color and its red, green, and blue intensities. The command

:save [*techStyle displayStyle monitorType*]

will save the current color map in a file named *techStyle.displayStyle.monitorType***.cmap**, where *techStyle* is the type of technology (e.g., **mos**), *displayStyle* is the kind of display specified by a **styletype** in the **style** section of a technology file (e.g., **7bit**), and *monitorType* is the type of the current monitor (e.g., **std**). If no arguments are given, the current technology style, display style, and monitor type are used. The command

:load [*techStyle displayStyle monitorType*]

will load the color map from the file named *techStyle.displayStyle.monitorType***.cmap** as above. If no arguments are given, the current technology style, display style, and monitor type are used. When loading color maps, Magic looks first in the current directory, then in the system library.

Magic Tutorial #6: Design-Rule Checking

John Ousterhout

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies

Commands introduced in this tutorial:

:drc

Macros introduced in this tutorial:

y

1 Continuous Design-Rule Checking

When you are editing a layout with Magic, the system automatically checks design rules on your behalf. Every time you paint or erase, and every time you move a cell or change an array structure, Magic rechecks the area you changed to be sure you haven't violated any of the layout rules. If you do violate rules, Magic will display little white dots in the vicinity of the violation. This error paint will stay around until you fix the problem; when the violation is corrected, the error paint will go away automatically. Error paint is written to disk with your cells and will re-appear the next time the cell is read in. There is no way to get rid of it except to fix the violation.

Continuous design-rule checking means that you always have an up-to-date picture of design-rule errors in your layout. There is never any need to run a massive check over the whole design unless you change your design rules. When you make small changes to an existing layout, you will find out immediately if you've introduced errors, without having to completely recheck the entire layout.

To see how the checker works, run Magic on the cell **tut6a**. This cell contains several areas of metal (blue), some of which are too close to each other or too narrow. Try painting and erasing metal to make the error paint go away and re-appear again.

2 Getting Information about Errors

In many cases, the reason for a design-rule violation will be obvious to you as soon as you see the error paint. However, Magic provides several commands for you to use to find violations and figure what's wrong in case it isn't obvious. All of the design-rule checking commands have the form

:drc *option*

where *option* selects one of several commands understood by the design-rule checker. If you're not sure why error paint has suddenly appeared, place the box around the error paint and invoke the command

:drc why

This command will recheck the area underneath the box, and print out the reasons for any violations that were found. You can also use the macro **y** to do the same thing. Try this on some of the errors in **tut6a**. It's a good idea to place the box right around the area of the error paint: **:drc why** rechecks the entire area under the box, so it may take a long time if the box is very large.

If you're working in a large cell, it may be hard to see the error paint. To help locate the errors, select a cell and then use the command

:drc find [*nth*]

If you don't provide the *nth* argument, the command will place the box around one of the errors in the selected cell, and print out the reason for the error, just as if you had typed **:drc why**. If you invoke the command repeatedly, it will step through all of the errors in the selected cell. (remember, the "." macro can be used to repeat the last long command; this will save you from having to retype **:drc find** over and over again). Try this out on the errors in **tut6a**. If you type a number for *nth*, the command will go to the *nth* error in the selected cell, instead of the next one. If you invoke this command with no cell selected, it searches the edit cell.

A third drc command is provided to give you summary information about errors in hierarchical designs. The command is

:drc count

This command will search every cell (visible or not) that lies underneath the box to see if any have errors in them. For each cell with errors, **:drc count** will print out a count of the number of error areas.

3 Errors in Hierarchical Layouts

The design-rule checker works on hierarchical layouts as well as single cells. There are three overall rules that describe the way that Magic checks hierarchical designs:

1. The paint in each cell must obey all the design rules by itself, without considering the paint in any other cells, including its children.
2. The combined paint of each cell and all of its descendants (subcells, sub-subcells, etc.) must be consistent. If a subcell interacts with paint or with other subcells in a way that introduces a design-rule violation, an error will appear in the parent. In designs with many levels of hierarchy, this rule is applied separately to each cell and its descendants.
3. Each array must be consistent by itself, without considering any other subcells or paint in its parent. If the neighboring elements of an array interact to produce a design-rule violation, the violation will appear in the parent.

To see some examples of interaction errors, edit the cell **tut6b**. This cell doesn't make sense electrically, but illustrates the features of the hierarchical checker. On the left are two subcells that are too close together. In addition, the subcells are too close to the red paint in the top-level cell. Move the subcells and/or modify the paint to make the errors go away and reappear. On the right side of **tut6b** is an array whose elements interact to produce a design-rule violation. Edit an element of the array to make the violation go away. When there are interaction errors between the elements of an array, the errors always appear near one of the four corner elements of the array (since the array spacing is uniform, Magic only checks interactions near the corners; if these elements are correct, all the ones in the middle must be correct too).

It's important to remember that each of the three overall rules must be satisfied independently. This may sometimes result in errors where it doesn't seem like there should be any. Edit the cell **tut6c** for some examples of this. On the left side of the cell there is a sliver of paint in the parent that extends paint in a subcell. Although the overall design is correct, the sliver of paint in the parent is not correct by itself, as required by the first overall rule above. On the right side of **tut6c** is an array with spacing violations between the array elements. Even though the paint in the parent masks some of the problems, the array is not consistent by itself so errors are flagged. The three overall rules are more conservative than strictly necessary, but they reduce the amount of rechecking Magic must do. For example, the array rule allows Magic to deduce the correctness of an array by looking only at the corner elements; if paint from the parent had to be considered in checking arrays, it would be necessary to check the entire array since there might be parent paint masking some errors but not all (as, for example, in **tut6c**).

Error paint appears in different cells in the hierarchy, depending on what kind of error was found. Errors resulting from paint in a single cell cause error paint to appear in that cell. Errors resulting from interactions and arrays appear in the parent of the interacting cells or array. Because of the way Magic makes interaction checks, errors can sometimes "bubble up" through the hierarchy and appear in multiple cells. When two cells overlap, Magic checks this area by copying all the paint in that area from both cells (and their descendants) into a buffer and then checking the buffer. Magic is unable to tell the difference between an error from one of the subcells and an error that comes about because the two subcells overlap incorrectly. This means that errors in an interaction area of a cell may also appear in the cell's parent. Fixing the error in the subcell will cause the error in the parent to go away also.

4 Turning the Checker Off

We hope that in most cases the checker will run so quickly and quietly that you hardly know it's there. However, there will probably be some situations where the checker is irksome. This section describes several ways to keep the checker out of your hair.

If you're working on a cell with lots of design-rule violations the constant redisplay caused by design-rule checking may get in your way more than it helps. This is particularly true if you're in the middle of a large series of changes and don't care about design-rule violations until the changes are finished. You can stop the redisplay using the command

:see no errors

After this command is typed, design-rule errors will no longer be displayed on the screen. The design-rule checker will continue to run and will store error information internally, but it won't bother you by displaying it on the screen. When you're ready to see errors again, type

:see errors

There can also be times when it's not the redisplay that's bothersome, but the amount of CPU time the checker takes to recheck what you've changed. For example, if a large subcell is moved to overlap another large subcell, the entire overlap area will have to be rechecked, and this could take several minutes. If the prompt on the text screen is a "]" character, it means that the command has completed but the checker hasn't caught up yet. You can invoke new commands while the checker is running, and the checker will suspend itself long enough to process the new commands.

If the checker takes too long to interrupt itself and respond to your commands, you have several options. First, you can hit the interrupt key (often ˆC) on the keyboard. This will stop the checker immediately and wait for your next command. As soon as you issue a command or push a mouse button, the checker will start up again. To turn the checker off altogether, type the command

:drc off

From this point on, the checker will not run. Magic will record the areas that need rechecking but won't do the rechecks. If you save your file and quit Magic, the information about areas to recheck will be saved on disk. The next time you read in the cell, Magic will recheck those areas, unless you've still got the checker turned off. There is nothing you can do to make Magic forget about areas to recheck; **:drc off** merely postpones the recheck operation to a later time.

Once you've turned the checker off, you have two ways to make sure everything has been rechecked. The first is to type the command

:drc catchup

This command will run the checker and wait until everything has been rechecked and errors are completely up to date. When the command completes, the checker will still be enabled or disabled just as it was before the command. If you get tired of waiting for **:drc catchup**, you can always hit the interrupt key to abort the command; the recheck areas will be remembered for later. To turn the checker back on permanently, invoke the command

:drc on

5 Porting Layouts from Other Systems

You should not need to read this section if you've created your layout from scratch using Magic or have read it from CIF using Magic's CIF or Calma reader. However, if you are bringing into Magic a layout that was created using a different editor or an old version of Magic that didn't have continuous checking, read on. You may also need to read this section if you've changed the design rules in the technology file.

In order to find out about errors in a design that wasn't created with Magic, you must force Magic to recheck everything in the design. Once this global recheck has been done, Magic will use its continuous checker to deal with any changes you make to the design; you should only need to do the global recheck once. To make the global recheck, load your design, place the box around the entire design, and type

:drc check

This will cause Magic to act as if the entire area under the box had just been modified: it will recheck that entire area. Furthermore, it will work its way down through the hierarchy; for every subcell found underneath the box, it will recheck that subcell over the area of the box.

If you get nervous that a design-rule violation might somehow have been missed, you can use **:drc check** to force any area to be rechecked at any time, even for cells that were created with Magic. However, this should never be necessary unless you've changed the design rules. If the number of errors in the layout ever changes because of a **:drc check**, it is a bug in Magic and you should notify us immediately.

Magic Tutorial #7: Netlists and Routing

John Ousterhout

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #3: Advanced Painting (Wiring and Plowing)
Magic Tutorial #4: Cell Hierarchies
Magic Tutorial #5: Multiple Windows

Netlist Commands introduced in this tutorial:

:extract, :flush, :ripup, :savenetlist, :trace, :writeall

Layout Commands introduced in this tutorial:

:channel, :route

Macros introduced in this tutorial:

(none)

1 Introduction

This tutorial describes how to use Magic's automatic routing tools to make interconnections between subcells in a design. In addition to the standard Magic router, which is invoked by the **route** command and covered in this tutorial, two other routing tools are available. A gate-array router *Garouter* permits user specified channel definitions, terminals in the interior of cells, and route-throughs across cells. To learn about the gate-array router read this first then "Magic Tutorial #12: Routing Gate Arrays". Finally Magic provides an interactive maze-router that takes graphic hints, the *Irouter*, that permits the user to control the overall path of routes while leaving the tedious details to Magic. The *Irouter* is documented in "Magic Tutorial #10: The Interactive Router".

The standard Magic router provides an *obstacle-avoidance* capability: if there is mask material in the routing areas, the router can work under, over, or around that material to complete the connections. This means that you can pre-route key signals by hand and have Magic route the less important signals automatically. In addition, you can route power and ground by hand (right now we don't have any power-ground routing tools, so you *have* to route them by hand).

The router *only* makes connections between subcells; to make point-to-point connections between pieces of layout within a single cell you should use the wiring command described in "Magic Tutorial #3: Advanced Painting (Wiring and Plowing) " or the maze router described in "Magic Tutorial #10: The Interactive Router". If you only need to make a few connections you are probably better off doing them manually.

The first step in routing is to tell Magic what should be connected to what. This information is contained in a file called a *netlist*. Sections 2, 3, 4, and 5 describe how to create and modify netlists using Magic's interactive netlist editing tools. Once you've created a netlist, the next step is to invoke the router. Section 6 shows how to do this, and gives a brief summary of what goes on inside the routing tools. Unless your design is very simple and has lots of free space, the routing probably won't succeed the first time. Section 7 describes the feedback provided by the routing tools. Sections 8 and 9 discuss how you can modify your design in light of this feedback to improve its routability. You'll probably need to iterate a few times until the routing is successful.

2 Terminals and Netlists

A netlist is a file that describes a set of desired connections. It contains one or more *nets*. Each net names a set of *terminals* that should all be wired together. A terminal is simply a label attached to a piece of mask material within a subcell; it is distinguishable from ordinary labels within a subcell by its presence within a netlist file and by certain characteristics common to terminals, as described below.

The first step in building a netlist is to label the terminals in your design. Figure 1 shows an example. Each label should be a line or rectangle running along the edge of the cell (point terminals are not allowed). The router will make a connection to the cell somewhere along a terminal's length. If the label isn't at the edge of the cell, Magic will route recklessly across the cell to reach the terminal, taking the shortest path between the terminal and a routing channel. It's almost always a good idea to arrange for terminal labels to be at cell edges. The label must be at least as wide as the minimum width of the routing material; the wider you make the label, the more flexibility you give the router to choose a good point to connect to the terminal.

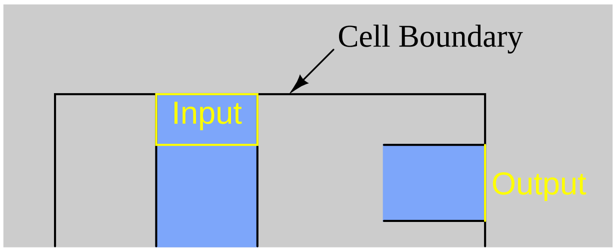


Figure 1: An example of terminal labels. Each terminal should be labeled with a line or rectangle along the edge of the cell.

Terminal labels must be attached to mask material that connects directly to one of Magic's two routing layers (Routing layers are defined in Magic's technology file). For example, in the SCMOS process where the routing layers are metal1 and metal2, diffusion may not be used as a terminal since neither of the routing layers will connect directly to it. On the other hand, a terminal may be attached to diffusion-metal1 contact, since the metal1 routing layer will connect properly to it. Terminals can have arbitrary names, except that they should not contain slashes ("/") or the substring "feedthrough", and should not end in "@", "$", or "^". See Tutorial #2 for a complete description of labeling conventions.

For an example of good and bad terminals, edit the cell **tut7a**. The cell doesn't make any electrical sense, but contains several good and bad terminals. All the terminals with names like **bad1** are incorrect or undesirable for one of the reasons given above, and those with names like **good4** are acceptable.

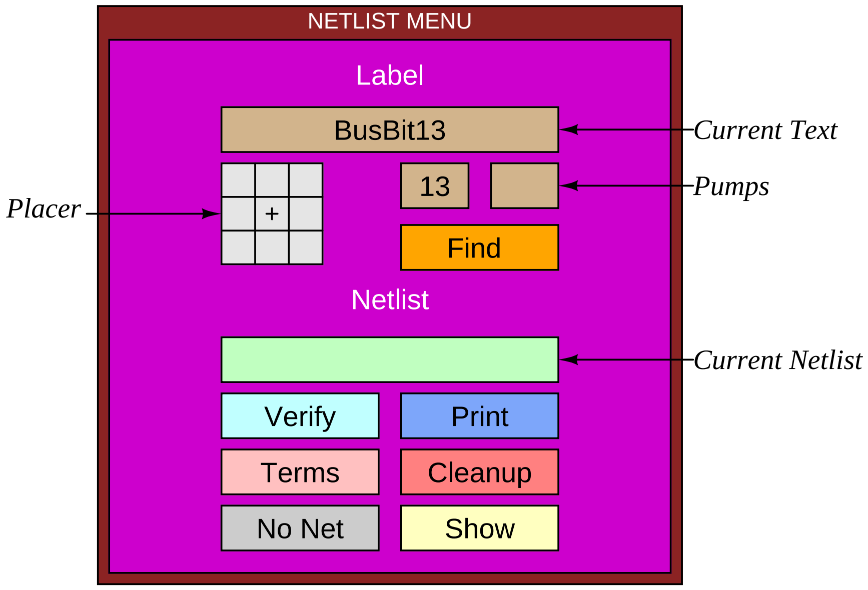


Figure 2: The netlist menu.

If you create two or more terminal labels with the same name in the same cell the router

Button	Action
Current Text	Left-click: prompt for more labels Right-click: advance to next label
Placer	Left-click: place label Right-click: change label text position
Pumps	Left-click: decrement number Right-click: increment number
Find	Search under box, highlight labels matching current text
Current Netlist	Left-click: prompt for new netlist name Right-click: use edit cell name as netlist name
Verify	Check that wiring matches netlist (same as typing **:verify** command)
Print	Print names of all terminals in selected net (same as typing **:print** command)
Terms	Place feedback areas on screen to identify all terminals in current netlist (same as **:showterms** command)
Cleanup	Check current netlist for missing labels and nets with less than two terminals (same as typing
 No Net	**:cleanup** command) Delete selected net (same as **:dnet** command)
Show	Highlight paint connected to material under box (same as typing **:shownet** command)

Table 1: A summary of all the netlist menu button actions.

will assume that they are electrically equivalent (connected together within the cell). Because of this, when routing the net it will feel free to connect to whichever one of the terminals is most convenient, and ignore the others. In some cases the router may take advantage of electrically equivalent terminals by using *feed throughs*: entering a cell at one terminal to make one connection, and exiting through an equivalent terminal on the way to make another connection for the same net.

3 Menu for Label Editing

Magic provides a special menu facility to assist you in placing terminal labels and editing netlists. To make the menu appear, invoke the Magic command

:specialopen netlist

A new window will appear in the lower-left corner of the screen, containing several rectangular areas on a purple background. Each of the rectangular areas is called a *button*. Clicking mouse buttons inside the menu buttons will invoke various commands to edit labels and netlists. Figure 2 shows a diagram of the netlist menu and Table I summarizes the meaning of button clicks in various menu items. The netlist menu can be grown, shrunk, and moved just like any other window; see “Magic Tutorial #5: Multiple Windows” for details. It also has its own private set of commands. To see what commands you can type in the netlist menu, move the cursor over the menu and type

:help

You shouldn’t need to type commands in the netlist menu very often, since almost everything you’ll need to do can be done using the menu. See Section 9 for a description of a few of the commands you can type; the complete set is described in the manual page *magic(1)*. One of the best uses for the commands is so that you can define macros for them and avoid having to go back and forth to the menu; look up the **:send** command in the man page to see how to do this. The top half of the menu is for placing labels and the bottom half is for editing netlists. This section describes the label facilities, and Section 4 describes the netlist facilities.

The label menu makes it easy for you to enter lots of labels, particularly when there are many labels that are the same except for a number, e.g. **bus1**, **bus2**, **bus3**, etc. There are four sections to the label menu: the current text, the placer, two pumps, and the **Find** button. To place labels, first click the left mouse button over the current text rectangle. Then type one or more labels on the keyboard, one per line. You can use this mechanism to enter several labels at once. Type return twice to signal the end of the list. At this point, the first of the labels you typed will appear in the current text rectangle.

To place a label, position the box over the area you want to label, then click the left mouse button inside one of the squares of the placer area. A label will be created with the current text. Where you click in the placer determines where the label text will appear relative to the label box: for example, clicking the left-center square causes the text to be centered just to the left of the box. You can place many copies of the same label by moving the box and clicking the placer area again. You can re-orient the text of a label by clicking the right mouse button inside the placer area. For example, if you would like to move a label’s text so that it appears centered above the label, place the box over the label and right-click the top-center placer square.

If you entered several labels at once, only the first appears in the current text area. However, you can advance to the next label by right-clicking inside the current text area. In this way you can place a long series of labels entirely with the mouse. Try using this mechanism to add labels to **tut7a**.

The two small buttons underneath the right side of the current text area are called pumps. To see how these work, enter a label name containing a number into the current text area, for example, **bus1**. When you do this, the “1” appears in the left pump. Right-clicking the pump causes the number to increment, and left-clicking the pump causes the number to decrement. This makes it easy for you to enter a series of numbered signal names. If a name has two numbers in it, the second number will appear in the second pump, and it can be incremented or decremented too. Try using the pumps to place a series of numbered names.

The last entry in the label portion of the menu is the **Find** button. This can be used to locate a label by searching for a given pattern. If you click the **Find** button, Magic will use the current text as a pattern and search the area underneath the box for a label whose name contains the pattern. Pattern-matching is done in the same way as in *csh*, using the special characters “*”, “?”, “
”, “[”, and “]”. Try this on **tut7a**: enter “good*” into the current text area, place the box around the whole cell, then click on the “Find” button. For each of the good labels, a feedback area will be

created with white stripes to highlight the area. The **:feedback find** command can be used to step through the areas, and **:feedback clear** will erase the feedback information from the screen. The **:feedback** command has many of the same options as **:drc** for getting information about feedback areas; see the Magic manual page for details, or type **:feedback help** for a synopsis of the options.

4 Netlist Editing

After placing terminal labels, the next step is to specify the connections between them; this is called netlist editing. The bottom half of the netlist menu is used for editing netlists. The first thing you must do is to specify the netlist you want to edit. Do this by clicking in the current netlist box. If you left-click, Magic will prompt you for the netlist name and you can type it at the keyboard. If you right-click, Magic will use the name of the edit cell as the current netlist name. In either case, Magic will read the netlist from disk if it exists and will create a new netlist if there isn't currently a netlist file with the given name. Netlist files are stored on disk with a ".net" extension, which is added by Magic when it reads and writes files. You can change the current netlist by clicking the current netlist button again. Startup Magic on the cell **tut7b**, open the netlist menu, and set the current netlist to **tut7b**. Then expand the subcells in **tut7b** so that you can see their terminals.

Button	Action
Left	Select net, using nearest terminal to cursor.
Right	Toggle nearest terminal into or out of current net.
Middle	Find nearest terminal, join its net with the current net.

Table 2: The actions of the mouse buttons when the terminal tool is in use.

Netlist editing is done with the netlist tool. If you haven't already read "Tutorial #3: Advanced Painting (Wiring and Plowing)", you should read it now, up through Section 2.1. Tutorial #3 explained how to change the current tool by using the space macro or by typing **:tool**. Switch tools to the netlist tool (the cursor will appear as a thick square).

When the netlist tool is in use the left, right, and middle buttons invoke select, toggle, and join operations respectively (see Table II). To see how they work, move the cursor over the terminal **right4** in the top subcell of **tut7b** and click the left mouse button (you may have to zoom in a bit to see the labels; terminals are numbered in clockwise order: **right4** is the fourth terminal from the top on the right side). This causes the net containing that terminal to be selected. Three hollow white squares will appear over the layout, marking the terminals that are supposed to be wired together into **right4**'s net. Left-click over the **left3** terminal in the same subcell to select its net, then select the **right4** net again.

The right button is used to toggle terminals into or out of the current net. If you right-click over a terminal that is in the current net, then it is removed from the current net. If you right-click over a terminal that isn't in the current net, it is added to the current net. A single terminal can only be in one net at a time, so if a terminal is already in a net when you toggle it into another net then Magic will remove it from the old net. Toggle the terminal **top4** in the bottom cell out of, then back into, the net containing **right4**. Now toggle **left3** in the bottom cell into this net. Magic warns you because it had to remove **left3** from another net in order to add it to **right4**'s net. Type **u** to undo this change, then left-click on **left3** to make sure it got restored to its old net by the undo. All of the netlist-editing operations are undo-able.

The middle button is used to merge two nets together. If you middle-click over a terminal, all the terminals in its net are added to the current net. Play around with the three buttons to edit the netlist **tut7b**.

Note: the router does not make connections to terminals in the top level cell. It only works with terminals in subcells, or sub-subcells, etc. Because of this, the netlist editor does not permit you to select terminals in the top level cell. If you click over such a terminal Magic prints an error message and refuses to make the selection.

If you left-click over a terminal that is not currently in a net, Magic creates a new net automatically. If you didn't really want to make a new net, you have several choices. Either you can toggle the terminal out of its own net, you can undo the select operation, or you can click the **No Net** button in the netlist menu (you can do this even while the cursor is in the square shape). The **No Net** button removes all terminals from the current net and destroys the net. It's a bad idea to leave single-net terminals in the netlist: the router will treat them as errors.

There are two ways to save netlists on disk; these are similar to the ways you can save layout cells. If you type

:savenetlist [*name*]

with the cursor over the netlist menu, the current netlist will be saved on disk in the file *name*.**net**. If no *name* is typed, the name of the current netlist is used. If you type the command

:writeall

then Magic will step through all the netlists that have been modified since they were last written, asking you if you'd like them to be written out. If you try to leave Magic without saving all the modified netlists, Magic will warn you and give you a chance to write them out.

If you make changes to a netlist and then decide you don't want them, you can use the **:flush** netlist command to throw away all of the changes and re-read the netlist from its disk file. If you create netlists using a text editor or some other program, you can use **:flush** after you've modified the netlist file in order to make sure that Magic is using the most up-to-date version.

The **Print** button in the netlist menu will print out on the text screen the names of all the terminals in the current net. Try this for some of the nets in **tut7b**. The official name of a terminal looks a lot like a Unix file name, consisting of a bunch of fields separated by slashes. Each field except the last is the id of a subcell, and the last field is the name of the terminal. These hierarchical names provide unique names for each terminal, even if the same terminal name is re-used in different cells or if there are multiple copies of the same cell.

The **Verify** button will check the paint of the edit cell to be sure it implements the connections specified in the current netlist. Feedback areas are created to show nets that are incomplete or nets that are shorted together.

The **Terms** button will cause Magic to generate a feedback area over each of the terminals in the current netlist, so that you can see which terminals are included in the netlist. If you type the command **:feedback clear** in a layout window then the feedback will be erased.

The **Cleanup** button is there as a convenience to help you cleanup your netlists. If you click on it, Magic will scan through the current netlist to make sure it is reasonable. **Cleanup** looks for two error conditions: terminal names that don't correspond to any labels in the design, and nets that don't have at least two terminals. When it finds either of these conditions it prints a message and gives you the chance to either delete the offending terminal (if you type **dterm**), delete the offending net (**dnet**), skip the current problem without modifying the netlist and continue looking for other problems (**skip**), or abort the **Cleanup** command without making any more changes (**abort**).

The **Show** button provides an additional mechanism for displaying the paint in the net. If you place the box over a piece of paint and click on **Show**, Magic will highlight all of the paint in the net under the box. This is similar to pointing at the net and typing **s** three times to select the net, except that **Show** doesn't select the net (it uses a different mechanism to highlight it), and **Show** will trace through all cells, expanded or not (the selection mechanism only considers paint in expanded cells). Once you've used **Show** to highlight a net, the only way to make the highlighting go away is to place the box over empty space and invoke **Show** again. **Show** is an old command that pre-dates the selection interface, but we've left it in Magic because some people find it useful.

5 Netlist Files

Netlists are stored on disk in ordinary text files. You are welcome to edit those files by hand or to write programs that generate the netlists automatically. For example, a netlist might be generated by a schematic editor or by a high-level simulator. See the manual page *net(5)* for a description of netlist file format.

6 Running the Router

Once you've created a netlist, it is relatively easy to invoke the router. First, place the box around the area you'd like Magic to consider for routing. No terminals outside this area will be considered, and Magic will not generate any paint more than a few units outside this area (Magic may use the next routing grid line outside the area). Load **tut7d**, **:flush** the netlist if you made any changes to it, set the box to the bounding box of the cell, and then invoke the router using the command:

```
:route
```

When the command completes, the netlist should be routed. Click the **Verify** netlist button to make sure the connections were made correctly. Try deleting a piece from one of the wires and verify again. Feedback areas should appear to indicate where the routing was incorrect. Use the **:feedback** command to step through the areas and, eventually, to delete the feedback (**:feedback help** gives a synopsis of the command options).

If the router is unable to complete the connections, it will report errors to you. Errors may be reported in several ways. For some errors, such as non-existent terminal names, messages will be printed. For other errors, cross-hatched feedback areas will be created. Most of the feedback areas have messages similar to "Net shifter/bit[0]/phi1: Can't make bottom connection." To see the message associated with a feedback area, place the box over the feedback area and type **:feedback**

why. In this case the message means that for some reason the router was unable to connect the specified net (named by one of its terminals) within one of the routing channel. The terms "bottom", "top", etc. may be misnomers because Magic sometimes rotates channels before routing: the names refer to the direction at the time the channel was routed, not the direction in the circuit. However, the location of the feedback area indicates where the connection was supposed to have been made.

You've probably noticed by now that the router sometimes generates unnecessary wiring, such as inserting extra jogs and U-shapes in wires (look next to **right3** in the top cell). These jogs are particularly noticeable in small examples. However, the router actually does *better* on larger examples: there will still be a bit of extra wire, but it's negligible in comparison to the total wire length on a large chip. Some of this wire is necessary and important: it helps the router to avoid several problem situations that would cause it to fail on more difficult examples. However, you can use the **straighten** command described in "Magic Tutorial #3: Advanced Painting (Wiring and Plowing)" to remove unnecessary jogs. Please don't judge the router by its behavior on small examples. On the other hand, if it does awful things on big examples, we'd like to know about it.

All of the wires placed by the router are of the same width, so the router won't be very useful for power and ground wiring.

When using the Magic router, you can wire power and ground by hand before running the router. The router will be able to work around your hand-placed connections to make the connections in the netlist. If there are certain key signals that you want to wire carefully by hand, you can do this too; the router will work around them. Signals that you route by hand should not be in the netlist. **Tutorial7b** has an example of "hand routing" in the form of a piece of metal in the middle of the circuit. Undo the routing, and try modifying the metal and/or adding more hand routing of your own to see how it affects the routing.

The Magic router has a number of options useful for getting information about the routing and setting routing parameters. You need to invoke the **route** command once for each option you want to specify; then type **:route** with no options to start up the router with whatever parameters you've set. The **viamin**, option which invokes a routing post-pass is, of course, invoked AFTER routing. Type **:route netlist** *file* to specify a netlist for the routing without having to open up the netlist menu. The **metal** option lets you toggle metal maximization on and off; if metal maximization is turned on, the router converts routing from the alternate routing layer ("poly") to the preferred routing layer ("metal") wherever possible. The **vias** option controls metal maximization by specifying how many grid units of "metal" conversion make it worthwhile to place vias; setting this to 5 means that metal maximization will add extra vias only if 5 or more grid units of "poly" can be converted to "metal". View the current technology's router parameters with the **tech** option. The **jog**, **obstacle**, and **steady** options let you view and change parameters to control the channel router (this feature is for advanced users). The **viamin** option invokes a via minimization algorithm which reduces the number of vias in a routed layout. This can be used as a post-processing step to improve the quality of the routing. This may be useful even when using another router to do the actual routing. Finally, show all parameter values with the **settings** option. The options and their actions are summarized in Table III.

Option	Action
end	Print the channel router end constant
end*real*	Set the channel router end constant
help	Print a summary of the router options
jog	Print the channel router minimum jog length
jog *int*	Set the minimum jog length, measured in grid units
metal	Toggle metal maximization on or off
netlist	Print the name of the current net list
netlist *file*	Set the current net list
obstacle	Print the channel router obstacle constant
obstacle *real*	Set the obstacle constant
settings	Print a list of all router parameters
steady	Print the channel router steady net constant
steady *int*	Set the steady net constant, measured in grid units
tech	Print router technology information
vias	Print the metal maximization via limit
vias *int*	Set the via limit
viamin	Minimize vias in a routed layout.

Table 3: A summary of all of Magic router options.

7 How the Router Works

In order to make the router produce the best possible results, it helps to know a little bit about how it works. The router runs in three stages, called *channel definition*, *global routing*, and *channel routing*. In the channel definition phase, Magic divides the area of the edit cell into rectangular routing areas called channels. The channels cover all the space under the box except the areas occupied by subcells. All of Magic's routing goes in the channel areas, except that stems (Section 8.2) may extend over subcells.

To see the channel structure that Magic chose, place the box in **tut7d** as if you were going to route, then type the command

:channel

in the layout window. Magic will compute the channel structure and display it on the screen as a collection of feedback areas. The channel structure is displayed as white rectangles. Type **:feedback clear** when you're through looking at them.

The second phase of routing is global routing. In the global routing phase, Magic considers each net in turn and chooses the sequence of channels the net must pass through in order to connect its terminals. The *crossing points* (places where the net crosses from one channel to another) are chosen at this point, but not the exact path through each channel.

In the third phase, each channel is considered separately. All the nets passing through that channel are examined at once, and the exact path of each net is decided. Once the routing paths have been determined, paint is added to the edit cell to implement the routing.

The router is grid-based: all wires are placed on a uniform grid. For the standard nMOS process the grid spacing is 7 units, and for the standard SCMOS process it is 8 units. If you type **:grid 8** after routing **tut7b**, you'll see that all of the routing lines up with its lower and left sides on grid lines. Fortunately, you don't have to make your cell terminals line up on even grid boundaries. During the routing Magic generates *stems* that connect your terminals up to grid lines at the edges of channels. Notice that there's space left by Magic between the subcells and the channels; this space is used by the stem generator.

8 What to do When the Router Fails

Don't be surprised if the router is unable to make all the connections the first time you try it on a large circuit. Unless you have extra routing space in your chip, you may have to make slight re-arrangements to help the router out. The paragraphs below describe things you can do to make life easier for the router. This section is not very well developed, so we'd like to hear about techniques you use to improve routability. If you discover new techniques, send us mail and we'll add them to this section.

8.1 Channel Structure

One of the first things to check when the router fails is the channel structure. If using the Magic router, type **:channel** to look at the channels. One common mistake is to have some of the desired routing area covered by subcells; Magic only runs wires where there are no subcells. Check to be sure that there are channels everywhere that you're expecting wires to run. If you place cells too close together, there may not be enough room to have a channel between the cells; when this happens Magic will route willy-nilly across the tops of cells to bring terminals out to channels, and will probably generate shorts or design-rule violations. To solve the problem, move the cells farther apart. If there are many skinny channels, it will be difficult for the router to produce good routing. Try to re-arrange the cell structure to line up edges of nearby cells so that there are as few channels as possible and they are as large as possible (before doing this you'll probably want to get rid of the existing routing by undo-ing or by flushing the edit cell).

8.2 Stems

Another problem has to do with the stem generator. Stems are the pieces of wiring that connect terminals up to grid points on the edges of channels. The current stem generation code doesn't know about connectivity or design rules. It simply finds the nearest routing grid point and wires out to that point, without considering any other terminals. If a terminal is not on the edge of the cell, the stem runs straight across the cell to the nearest channel, without any consideration for other material in the cell. If two terminals are too close together, Magic may decide to route them both to the same grid point. When this happens, you have two choices. Either you can move the cell so that the terminals have different nearest grid points (for example, you can line its terminals up with the grid lines), or if this doesn't work you'll have to modify the cell to make the terminals farther apart.

The place where stems cause the most trouble is in PLAs, many of which have been optimized to space the outputs as closely together as possible. In some cases the outputs are closer together than the routing grid, which is an impossible situation for the stem generator. In this case, we think the best approach is to change the PLA templates to space the outputs farther apart. Either space them exactly the same as the router grid (in which case you can line the PLAs up before routing so the terminals are already on the grid), or space the outputs at least 1.5 grid units apart so the stem generator won't have troubles. Having tightly-spaced PLA outputs is false economy: it makes it more difficult to design the PLAs and results in awful routing problems. Even if Magic could river-route out from tightly-spaced terminals to grid lines (which it can't), it would require N^2 space to route out N lines; it takes less area to stretch the PLA.

8.3 Obstacles

The router tends to have special difficulties with obstacles running along the edges of channels. When you've placed a power wire or other hand-routing along the edge of a channel, the channel router will often run material under your wiring in the other routing layer, thereby blocking both routing layers and making it impossible to complete the routing. Where this occurs, you can increase the chances of successful routing by moving the hand-routing away from the channel edges. It's especially important to keep hand-routing away from terminals. The stem generator will not pay any attention to hand-routing when it generates stems (it just makes a bee-line for the nearest grid point), so it may accidentally short a terminal to nearby hand-routing.

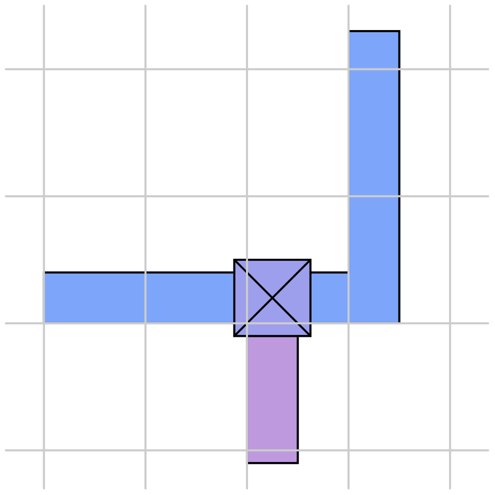


Figure 3: When placing hand routing, it is best to place wires with their left and bottom edges along grid lines, and contacts centered on the wires. In this fashion, the hand routing will block as few routing grid lines as possible.

When placing hand-routing, you can get better routing results by following the advice illustrated in Figure 3. First, display the routing grid. For example, if the router is using a 8-unit grid (which is true for the standard SCMOS technology), type **:grid 8**. Then place all your hand routing with its left and bottom edges along the grid lines. Because of the way the routing tools work, this approach results in the least possible amount of lost routing space.

9 More Netlist Commands

In addition to the netlist menu buttons and commands described in Section 4, there are a number of other netlist commands you can invoke by typing in the netlist window. Many of these commands are textual equivalents of the menu buttons. However, they allow you to deal with terminals by typing the hierarchical name of the terminal rather than by pointing to it. If you don't know where a terminal is, or if you have deleted a label from your design so that there's nothing to point to, you'll have to use the textual commands. Commands that don't just duplicate menu buttons are described below; see the *magic(1)* manual page for details on the others.

The netlist command

:extract

will generate a net from existing wiring. It looks under the box for paint, then traces out all the material in the edit cell that is connected electrically to that paint. Wherever the material touches subcells it looks for terminals in the subcells, and all the terminals it finds are placed into a new net. Warning: there is also an **extract** command for layout windows, and it is totally different from the **extract** command in netlist windows. Make sure you've got the cursor over the netlist window when you invoke this command!

The netlist editor provides two commands for ripping up existing routing (or other material). They are

:ripup
:ripup netlist

The first command starts by finding any paint in the edit cell that lies underneath the box. It then works outward from that paint to find all paint in the edit cell that is electrically connected to the starting paint. All of this paint is erased. (**:ripup** isn't really necessary, since the same effect can be achieved by selecting all the paint in the net and deleting the selection; it's a hangover from olden days when there was no selection). The second form of the command, **:ripup netlist**, is similar to the first except that it starts from each of the terminals in the current netlist instead of the box. Any paint in the edit cell that is electrically connected to a terminal is erased. The **:ripup netlist** command may be useful to ripup existing routing before rerouting.

The command

:trace [*name*]

provides an additional facility for examining router feedback. It highlights all paint connected to each terminal in the net containing *name*, much as the **Show** menu button does for paint connected to anything under the box. The net to be highlighted may be specified by naming one of its terminals, for example, **:trace shifter/bit[0]/phi1**. Use the trace command in conjunction with the nets specified in router feedback to see the partially completed wiring for a net. Where no net is specified, the **:trace** command highlights the currently selected net.

Magic Tutorial #8: Circuit Extraction

Walter Scott

Special Studies Program
Lawrence Livermore National Laboratory
P.O. Box 808, L-270
Livermore, CA 94550

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies

Commands introduced in this tutorial:

:extract

Macros introduced in this tutorial:

(none)

Changes since Magic version 4:

New form of **:extract unique**
Path length extraction with **:extract length**
Accurate resistance extraction with **:extresis**
Extraction of well connectivity and substrate nodes
Checking for global net connectedness in *ext2sim* (1)
New programs: *ext2spice* (1) and *extcheck* (1)

1 Introduction

This tutorial covers the use of Magic's circuit extractor. The extractor computes from the layout the information needed to run simulation tools such as *crystal* (1) and *esim* (1). This information includes the sizes and shapes of transistors, and the connectivity, resistance, and parasitic capacitance of nodes. Both capacitance to substrate and several kinds of internodal coupling capacitances are extracted.

Magic's extractor is both incremental and hierarchical: only part of the entire layout must be re-extracted after each change, and the structure of the extracted circuit parallels the structure of the layout being extracted. The extractor produces a separate **.ext** file for each **.mag** file in a hierarchical design. This is in contrast to previous extractors, such as Mextra, which produces a single **.sim** file that represents the flattened (fully-instantiated) layout.

Sections 2 through 4 introduce Magic's **:extract** command and some of its more advanced features. Section 5 describes what information actually gets extracted, and discusses limitations and inaccuracies. Section 6 talks about extraction styles. Although the hierarchical *ext* (5) format fully describes the circuit implemented by a layout, very few tools currently accept it. It is normally necessary to flatten the extracted circuit using one of the programs discussed in Section 7, such as *ext2sim* (1), *ext2spice* (1), or *extcheck* (1).

2 Basic Extraction

You can use Magic's extractor in one of several ways. Normally it is not necessary to extract all cells in a layout. To extract only those cells that have changed since they were extracted, use:

:load *root*
:extract

The extractor looks for a **.ext** file for every cell in the tree that descends from the cell *root*. The **.ext** file is searched for in the same directory that contains the cell's **.mag** file. Any cells that have been modified since they were last extracted, and all of their parents, are re-extracted. Cells having no **.ext** files are also re-extracted.

To try out the extractor on an example, copy all the **tut8***x* cells to your current directory with the following shell commands:

```
cp ~cad/lib/magic/tutorial/tut8*.mag .
```

Start magic on the cell **tut8a** and type **:extract**. Magic will print the name of each cell (**tut8a**, **tut8b**, **tut8c**, and **tut8d**) as it is extracted. Now type **:extract** a second time. This time nothing gets printed, since Magic didn't have to re-extract any cells. Now delete the piece of poly labelled "**delete me**" and type **:extract** again. This time, only the cell **tut8a** is extracted as it is the only one that changed. If you make a change to cell **tut8b** (do it) and then extract again, both **tut8b** and **tut8a** will be re-extracted, since **tut8a** is the parent of **tut8b**.

To force all cells in the subtree rooted at cell *root* to be re-extracted, use **:extract all**:

:load*root*
:extract all

Try this also on **tut8a**.

You can also use the **:extract** command to extract a single cell as follows:

:extract cell *name*

will extract just the selected (current) cell, and place the output in the file *name*. Select the cell **tut8b** (**tut8b_0**) and type **:extract cell differentFile** to try this out. After this command, the file **differentFile.ext** will contain the extracted circuit for the cell **tut8b**. The children of **tut8b** (in this case, the single cell **tut8d**) will not be re-extracted by this command. If more than one cell is selected, the upper-leftmost one is extracted.

You should be careful about using **:extract cell**, since even though you may only make a change to a child cell, all of its parents may have to be re-extracted. To re-extract all of the parents of the selected cell, you may use

:extract parents

Try this out with **tut8b** still selected. Magic will extract only the cell **tut8a**, since it is the only one that uses the cell **tut8b**. To see what cells would be extracted by **:extract parents** without actually extracting them, use

:extract showparents

Try this command as well.

3 Feedback: Errors and Warnings

When the extractor encounters problems, it leaves feedback in the form of stippled white rectangular areas on the screen. Each area covers the portion of the layout that caused the error. Each area also has an error message associated with it, which you can see by using the **:feedback** command. Type **:feedback help** while in Magic for assistance in using the **:feedback** command.

The extractor will always report extraction *errors*. These are problems in the layout that may cause the output of the extractor to be incorrect. The layout should be fixed to eliminate extraction errors before attempting to simulate the circuit; otherwise, the results of the simulation may not reflect reality.

Extraction errors can come from violations of transistor rules. There are two rules about the formation of transistors: no transistor can be formed, and none can be destroyed, as a result of cell overlaps. For example, it is illegal to have poly in one cell overlap diffusion in another cell, as that would form a transistor in the parent where none was present in either child. It is also illegal to have a buried contact in one cell overlap a transistor in another, as this would destroy the transistor. Violating these transistor rules will cause design-rule violations as well as extraction errors. These errors only relate to circuit extraction: the fabricated circuit may still work; it just won't be extracted correctly.

In general, it is an error for material of two types on the same plane to overlap or abut if they don't connect to each other. For example, in CMOS it is illegal for p-diffusion and n-diffusion to overlap or abut.

In addition to errors, the extractor can give *warnings*. If only warnings are present, the extracted circuit can still be simulated. By default, only some types of warnings are reported and displayed as feedback. To cause all warnings to be displayed, use

:extract warn all

The command

:extract warn *warning*

may be used to enable specific warnings selectively; see below. To cause no warnings to be displayed, or to disable display of a particular *warning*, use respectively

:extract warn no all or
:extract warn no *warning*

Three different kinds of warnings are generated. The **dup** warning checks to see whether you have two electrically unconnected nodes in the same cell labelled with the same name. If so, you are warned because the two unconnected nodes will appear to be connected in the resulting **.ext** file, which means that the extracted circuit would not represent the actual layout. This is bad if you're simulating the circuit to see if it will work correctly: the simulator will think the two nodes are connected, but since there's no physical wire between them, the electrons won't! When two unconnected nodes share the same label (name), the extractor leaves feedback squares over each instance of the shared name.

It's an excellent idea to avoid labelling two unconnected nodes with the same name within a cell. Instead, use the ”correct” name for one of the nodes, and some mnemonic but textually distinct name for the other nodes. For example, in a cell with multiple power rails, you might use **Vdd!** for one of the rails, and names like **Vdd#1** for the others. As an example, load the cell **tut8e**. If the two nodes are connected in a higher-level cell they will eventually be merged when the extracted circuit is flattened. If you want to simulate a cell out of context, but still want the higher-level nodes to be hooked up, you can always create a dummy parent cell that hooks them together, either with wire or by using the same name for pieces of paint that lie over the terminals to be connected; see the cell **tut8f** for an example of this latter technique.

You can use the command

:extract unique

as an automatic means of labelling nodes in the manner described above. Run this command on the cell **tut8g**. A second version of this command is provided for compatibility with previous versions of Magic. Running

:extract unique #

will only append a unique numeric suffix to labels that end with a “#”. Any other duplicate nodenames that also don't end in a “**!**” (the global nodename suffix as described in Section 5) are flagged by feedback.

A second type of warning, **fets**, checks to see whether any transistors have fewer diffusion terminals than the minimum for their types. For example, the transistor type "**dfet**" is defined in the **nmos** technology file as requiring two diffusion terminals: a source and a drain. If a capacitor with only one diffusion terminal is desired in this technology, the type **dcap** should be used instead. The **fets** warning is a consistency check for transistors whose diffusion terminals have been accidentally shorted together, or for transistors with insufficiently many diffusion terminals.

The third warning, **labels**, is generated if you violate the following guideline for placement of labels: Whenever geometry from two subcells abuts or overlaps, it's a good idea to make sure that there is a label attached to the geometry in each subcell *in the area of the overlap or along the line of abutment*. Following this guideline isn't necessary for the extractor to work, but it will result in noticeably faster extraction.

By default, the **dup** and **fets** warnings are enabled, and the **labels** warning is disabled.

Load the cell **tut8h**, expand all its children (**tut8i** and **tut8j**), and enable all extractor warnings with **:extract warn all**. Now extract **tut8h** and all of its children with **:extract**, and examine the feedback for examples of fatal errors and warnings.

4 Advanced Circuit Extraction

4.1 Lengths

The Magic extractor has a rudimentary ability to compute wire lengths between specific named points in a circuit. This feature is intended for use with technologies where the wire length between two points is more important than the total capacitance on the net; this may occur, for example, when extracting circuits with very long wires being driven at high speeds (*e.g.*, bipolar circuits). Currently, you must indicate to Magic which pairs of points are to have distances computed. You do this by providing two lists: one of *drivers* and one of *receivers*. The extractor computes the distance between each driver and each receiver that it is connected to.

Load the cell **tut8k**. There are five labels: two are drivers (**driver1** and **driver2**) and three are receivers (**receiverA**, **receiverB**, and **receiverC**). Type the commands:

```
:extract length driver driver1 driver2
:extract length receiver receiverA receiverB receiverC
```

Now enable extraction of lengths with **:extract do length** and then extract the cell (**:extract**). If you examine **tut8k.ext**, you will see several **distance** lines, corresponding to the driver-receiver distances described above. These distances are through the centerlines of wires connecting the two labels; where multiple paths exist, the shortest is used.

Normally the driver and receiver tables will be built by using **:source** to read a file of **:extract length driver** and **:extract length receiver** commands. Once these tables are created in Magic, they remain until you leave Magic or type the command

```
:extract length clear
```

which wipes out both tables.

Because extraction of wire lengths is *not* performed hierarchically, it should only be done in the root cell of a design. Also, because it's not hierarchical, it can take a long time for long, complex wires such as power and ground nets. This feature is still experimental and subject to change.

4.2 Resistance

Magic provides for more accurate resistance extraction using the **:extresis** command. **:extresis** provides a detailed resistance/capacitance description for nets where parasitic resistance is likely to significantly affect circuit timing.

4.2.1 Tutorial Introduction

To try out the resistance extractor, load in the cell **tut8r**. Extract it using **:extract**, pause magic, and run ext2sim on the cell with the command

```
ext2sim tut8r
```

This should produce **tut8r.sim**, **tut8r.nodes**, and **tut8r.al**. Restart magic and type

```
:extresis tolerance 10
:extresis
```

This will extract interconnect resistances for any net where the interconnect delay is at least one-tenth of the transistor delay. Magic should give the messages:

```
:extresis tolerance 10
:extresis
Adding net2; Tnew = 0.428038ns,Told = 0.3798ns
Adding net1; Tnew = 0.529005ns,Told = 0.4122ns
Total Nets: 7
Nets extracted: 2 (0.285714)
Nets output: 2 (0.285714)
```

These may vary slightly depending on your technology parameters. The **Adding [net]** lines describe which networks for which magic produced resistor networks. **Tnew** is the estimated delay on the net including the resistor parasitics, while **Told** is the delay without parasitics. The next line describes where magic thinks the slowest node in the net is. The final 3 lines give a brief summary of the total number of nets, the nets requiring extraction, and the number for which resistors were added to the output.

Running the resistance extractor also produced the file **cell.res.ext**. To produce a **.sim** file containing resistors, quit magic and type:

```
cat tut8r.ext tut8r.res.ext >tut8r.2.ext
ext2sim -R -t! -t# tut8r.2
```

Comparing the two files, **tut8r.sim** and **tut8r.2.sim**, shows that the latter has the nodes net1 and net2 split into several parts, with resistors added to connect the new nodes together.

4.2.2 General Notes on using the resistance extractor

To use **:extresis**, the circuit must first be extracted using **:extract** and flattened using ext2sim. When ext2sim is run, do not use the **-t#** and **-t!** flags (i.e. don't trim the trailing "#" and "!" characters) or the **-R** flag because **:extresis** needs the **.sim** and **.ext** names to correspond exactly, and it needs the lumped resistance values that the extractor produces. Also, do not delete or rename the **.nodes** file; **:extresis** needs this to run. Once the **.sim** and **.nodes** files have been produced, type the command **:extresis** while running magic on the root cell. As the resistance extractor runs, it will identify which nets (if any) for which it is producing RC networks, and will identify what it thinks is the "slowest" point in the network. When it completes, it will print a brief summary of how many nets it extracted and how many required supplemental networks. The resistance networks are placed in the file **root.res.ext**. To produce a **.sim** file with the supplemental resistors, type **cat root.ext root.res.ext** >**newname.ext**, and then rerun **ext2sim** on the new file. During this second **ext2sim** run, the **-t** flag may be used.

Like extraction of wire lengths, resistance extraction is *not* performed hierarchically; it should only be done in the root cell of a design and can take a long time for complex wires.

4.2.3 Options, Features, Caveats and Bugs

The following is a list of command line options and the arguments that they take.

- **tolerance [value]**
  This controls how large the resistance in a network must be before it is added to the output description. **value** is defined as the minimum ratio of transistor resistance to interconnect resistance that requires a resistance network. The default value is 1; values less than 1 will cause fewer resistors to be output and will make the program run faster, while values greater than 1 will produce more a larger, more accurate description but will run slower.

- **all**
  Causes all nets in the circuit to be extracted; no comparison between transistor size and lumped resistance is performed. This option is not recommended for large designs.

- **simplify [on/off]**
  Turns on/off the resistance network simplification routines. Magic normally simplifies the resistance network it extracts by removing small resistors; specifying this flag turns this feature off.

- **extout [on/off]**
  Turns on and off the writing of the `root.res.ext` file. The default value is on.

- **lumped [on/off]**
  Turns on the writing of `root.res.lump`. This file contains an updated value of the lumped resistance for each net that **:extresis** extracts.

- **silent [on/off]**
  This option suppresses printing of the name and location of nets for which resistors are produced.

- **skip mask**
  Specifies a list of layers that the resistance extractor is to ignore.

- **help**
  Print brief list of options.

Attribute labels may also be used to specify certain extractor options. For a description of attributes and how they work, see tutorial 2. Following is a description of **:extresis** attributes.

- **res:skip@**
  Causes this net to be skipped. This is useful for avoiding extraction of power supplies or other DC signals that are not labeled Vdd or GND.

- **res:force@**
  Forces extraction of this net regardless of its lumped resistance value. Nets with both skip and force labels attached will cause the extractor to complain.

- **res:min=[value]@**
  Sets the smallest resistor size for this net. The default value is the resistance of the largest driving transistor divided by the tolerance described above.

- **res:drive@** - Nets with no driving transistors will normally not be extracted. This option allows the designer to specify from where in the net the signal is driven. This is primarily useful when extracting subcells, where the transistors driving a given signal may be located in a different cell.

4.2.4 Technology File Changes

Certain changes must be made in the extract section of the technology file to support resistance extraction. These include the **fetresist** and **contact** lines, plus a small change to the fet line. Full details can be found in Magic Maintainer's Manual #2. The only thing to note is that, contrary to the documentation, the **gccap** and **gscap** parts of the fet line MUST be set; the resistance extractor uses them to calculate RC time constants for the circuit.

5 Extraction Details and Limitations

This section explores in greater depth what gets extracted by Magic, as well as the limitations of the circuit extractor. A detailed explanation of the format of the **.ext** files output by Magic may be found in the manual page *ext* (5). "Magic Maintainer's Manual #2: The Technology File" describes how extraction parameters are specified for the extractor.

5.1 Nodes

Magic approximates the pieces of interconnect between transistors as "nodes". A node is like an equipotential region, but also includes a lumped resistance and capacitance to substrate. Figure 1

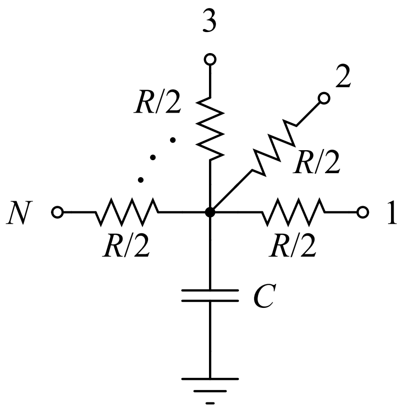


Figure 1: Each node extracted by Magic has a lumped resistance *R* and a lumped capacitance *C* to the substrate. These lumped values can be interpreted as in the diagram above, in which each device connected to the node is attached to one of the points *1*, *2*, . . . , *N*.

shows how these lumped values are intended to be interpreted by the analysis programs that use the extracted circuit.

Each node in an extracted circuit has a name, which is either one of the labels attached to the geometry in the node if any exist, or automatically generated by the extractor. These latter names are always of the form *p_x_y#*, where *p*, *x*, and *y* are integers, *e.g.*, **3_104_17#**. If a label ending in the character "**!**" is attached to a node, the node is considered to be a "global". Post-processing programs such as *ext2sim* (1) will check to ensure that nodes in different cells that are labelled with the same global name are electrically connected.

Nodes may have attributes attached to them as well as names. Node attributes are labels ending in the special character "**@**", and provide a mechanism for passing information to analysis programs such as *crystal* (1). The man page *ext* (5) provides additional information about node attributes.

5.2 Resistance

Magic extracts a lumped resistance for each node, rather than a point-to-point resistance between each pair of devices connected to that node. The result is that all such point-to-point resistances are approximated by the worst-case resistance between any two points in that node.

By default, node resistances are approximated rather than computed exactly. For a node comprised entirely of a single type of material, Magic will compute the node's total perimeter and area. It then solves a quadratic equation to find the width and height of a simple rectangle with this same perimeter and area, and approximates the resistance of the node as the resistance of this "equivalent" rectangle. The resistance is always taken in the longer dimension of the rectangle. When a node contains more than a single type of material, Magic computes an equivalent rectangle for each type, and then sums the resistances as though the rectangles were laid end-to-end.

This approximation for resistance does not take into account any branching, so it can be significantly in error for nodes that have side branches. Figure 2 gives an example. For global signal trees such as clocks or power, Magic's estimate of resistance will likely be several times higher than the actual resistance between two points.

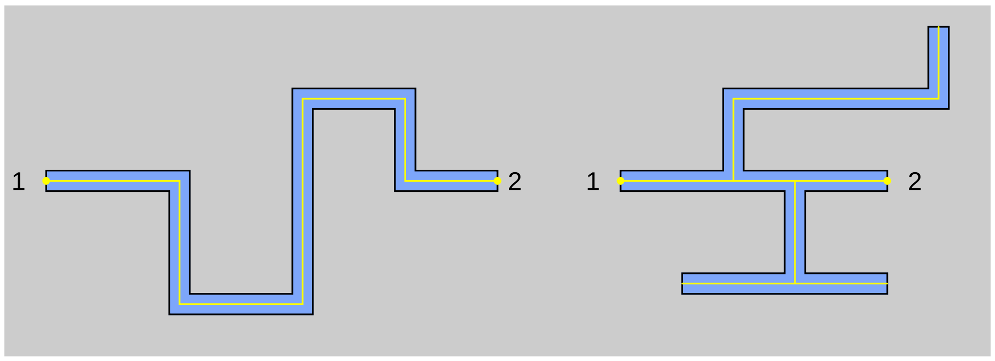


Figure 2: Magic approximates the resistance of a node by assuming that it is a simple wire. The length and width of the wire are estimated from the node's perimeter and area. (a) For non-branching nodes, this approximation is a good one. (b) The computed resistance for this node is the same as for (a) because the side branches are counted, yet the actual resistance between points 1 and 2 is significantly less than in (a).

The approximated resistance also does not lend itself well to hierarchical adjustments, as does capacitance. To allow programs like **ext2sim** to incorporate hierarchical adjustments into a resistance approximation, each node in the **.ext** file also contains a perimeter and area for each "resistance class" that was defined in the technology file (see "Maintainer's Manual #2: The Technology File," and *ext* (5)). When flattening a circuit, **ext2sim** uses this information along with adjustments to perimeter and area to produce the value it actually uses for node resistance.

If you wish to disable the extraction of resistances and node perimeters and areas, use the command

:extract no resistance

which will cause all node resistances, perimeters, and areas in the **.ext** file to be zero. To re-enable extraction of resistance, use the command

:extract do resistance.

Sometimes it's important that resistances be computed more accurately than is possible using the lumped approximation above. Magic's **:extresist** command does this by computing explicit two-terminal resistors and modifying the circuit network to include them so it reflects more exactly the topology of the layout. See the section on **Advanced Extraction** for more details on explicit resistance extraction with **:extresist**.

5.3 Capacitance

Capacitance to substrate comes from two different sources. Each type of material has a capacitance to substrate per unit area. Each type of edge (i.e, each pair of types) has a capacitance to substrate per unit length. See Figure 3. The computation of capacitance may be disabled with

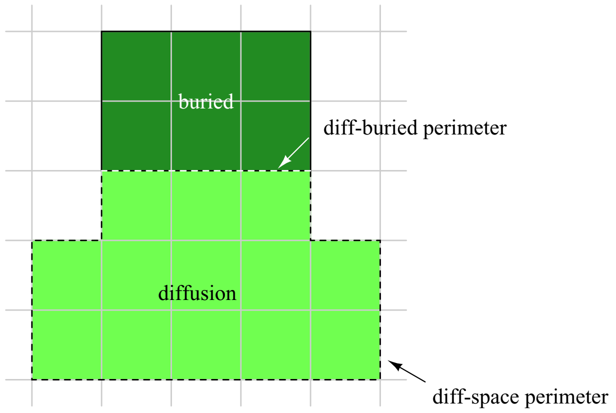


Figure 3: Each type of edge has capacitance to substrate per unit length. Here, the diffusion-space perimeter of 13 units has one value per unit length, and the diffusion-buried perimeter of 3 units another. In addition, each type of material has capacitance per unit area.

:extract no capacitance

which causes all substrate capacitance values in the **.ext** file to be zero. It may be re-enabled with

:extract do capacitance.

Internodal capacitance comes from three sources, as shown in Figure 4. When materials of two different types overlap, the capacitance to substrate of the one on top (as determined by the technology) is replaced by an internodal capacitance to the one on the bottom. Its computation may be disabled with

:extract no coupling

which will also cause the extractor to run 30% to 50% faster. Extraction of coupling capacitances can be re-enabled with

:extract do coupling.

Whenever material from two subcells overlaps or abuts, the extractor computes adjustments to substrate capacitance, coupling capacitance, and node perimeter and area. Often, these adjustments make little difference to the type of analysis you are performing, as when you wish only to compare netlists. Even when running Crystal for timing analysis, the adjustments can make less than a 5% difference in the timing of critical paths in designs with only a small amount of inter-cell overlap. To disable the computation of these adjustments, use

:extract no adjustment

which will result in approximately 50% faster extraction. This speedup is not entirely additive with the speedup resulting from **:extract no coupling**. To re-enable computation of adjustments, use **:extract do adjustment**.

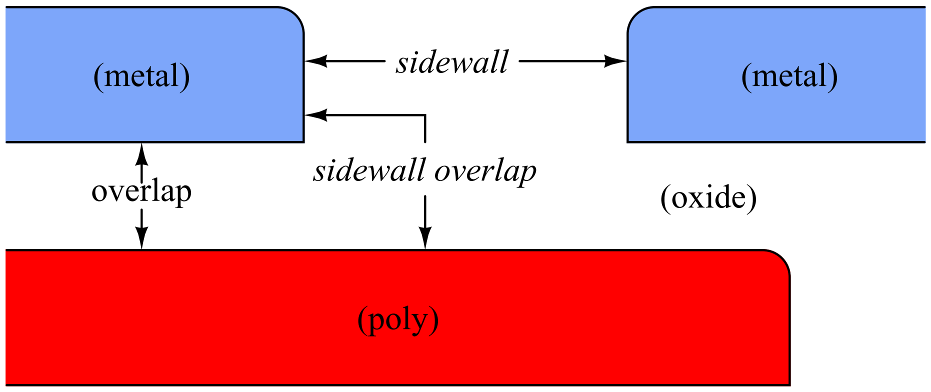


Figure 4: Magic extracts three kinds of internodal coupling capacitance. This figure is a cross-section (side view, not a top view) of a set of masks that shows all three kinds of capacitance. *Overlap* capacitance is parallel-plate capacitance between two different kinds of material when they overlap. *Sidewall* capacitance is parallel-plate capacitance between the vertical edges of two pieces of the same kind of material. *Sidewall overlap* capacitance is orthogonal-plate capacitance between the vertical edge of one piece of material and the horizontal surface of another piece of material that overlaps the first edge.

5.4 Transistors

Like the resistances of nodes, the lengths and widths of transistors are approximated. Magic computes the contribution to the total perimeter by each of the terminals of the transistor. See Figure 5. For rectangular transistors, this yields an exact L / W. For non-branching, non-rectangular transistors, it is still possible to approximate L / W fairly well, but substantial inaccuracies can be introduced if the channel of a transistor contains branches. Since most transistors are rectangular, however, Magic's approximation works well in practice.

Type	Loc	A P	Subs	Gate	Source	Drain
fet nfet	59 1 60 2	8 12	GND!	Mid2 4 **N3**	Out 4 0	Vss#0 4 0
fet nfet	36 1 37 2	8 12	Float	Mid1 4 **N2**	Mid2 4 0	Vss#0 4 0
fet nfet	4 1 5 2	8 12	Vss#0	In 4 **N1**	Mid1 4 0	Vss#0 4 0
fet pfet	59 25 60 26	8 12	Vdd!	Mid2 4 **P3**	Vdd#0 4 0	Out 4 0
fet pfet	36 25 37 26	8 12	VBias	Mid1 4 **P2**	Vdd#0 4 0	Mid2 4 0
fet pfet	4 25 5 26	8 12	Vdd#0	In 4 **P1**	Vdd#0 4 0	Mid1 4 0

Table 1: The transistor section of **tut8l.ext**.

In addition to having gate, source, and drain terminals, MOSFET transistors also have a substrate terminal. By default, this terminal is connected to a global node that depends on the transistor's type. For example, p-channel transistors might have a substrate terminal of **Vdd!**, while n-channel transistors would have one of **GND!**. However, when a transistor is surrounded by explicit "well" material (as defined in the technology file), Magic will override the default substrate terminal with the node to which the well material is connected. This has several advantages: it al-

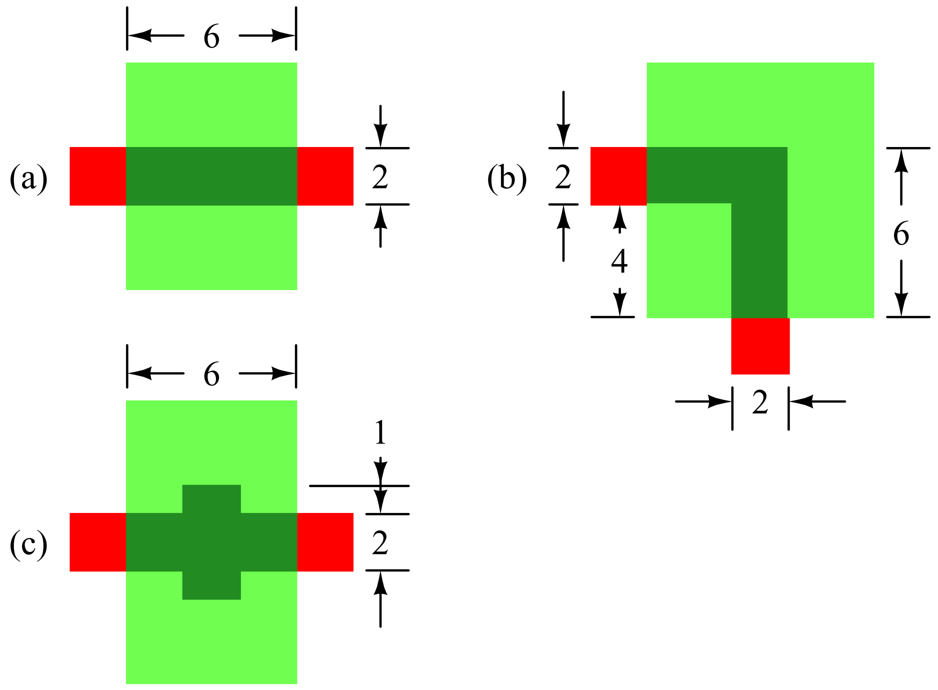


Figure 5: (a) When transistors are rectangular, it is possible to compute L/W exactly. Here *gateperim*$=4$, *srcperim*$=6$, *drainperim*$=6$, and $L/W = 2/6$. (b) The L/W of non-branching transistors can be approximated. Here *gateperim*$=4$, *srcperim*$=6$, *drainperim*$=10$. By averaging *srcperim* and *drainperim* we get $L/W = 2/8$. (c) The L/W of branching transistors is not well approximated. Here *gateperim*$=16$, *srcperim*$=2$, *drainperim*$=2$. Magic's estimate of L/W is $8/2$, whereas in fact because of current spreading, W is effectively larger than 2 and L effectively smaller than 8, so L/W is overestimated.

lows simulation of analog circuits in which wells are biased to different potentials, and it provides a form of checking to ensure that wells in a CMOS process are explicitly tied to the appropriate DC voltage.

Transistor substrate nodes are discovered by the extractor only if the transistor and the overlapping well layer are in the same cell. If they appear in different cells, the transistor's substrate terminal will be set to the default for the type of transistor.

Load the cell **tut8l**, extract it, and look at the file **tut8l.ext**. Table 1 shows the lines for the six transistors in the file. You'll notice that the substrate terminals (the *Subs* column) for all transistors are different. Since each transistor in this design has a different gate attribute attached to it (shown in bold in the table, *e.g.*, **N1**, **P2**, etc), we'll use them in the following discussion.

The simplest two transistors are **N3** and **P3**, which don't appear in any explicitly drawn wells. The substrate terminals for these are **GND!** and **Vdd!** respectively, since that's what the technology file says is the default for the two types of transistors. **N1** and **P1** are standard transistors that lie in wells tied to the ground and power rails, labelled in this cell as **Vss#0** and **Vdd#0** respectively. (They're not labelled **GND!** and **Vdd!** so you'll see the difference between **N1** and **N3**). **P2** lies in a well that is tied to a different bias voltage, **VBias**, such as might occur in an analog design. Finally, **N2** is in a well that isn't tied to any wire. The substrate node appears as **Float** because that's the label that was attached to the well surrounding **N2**.

The ability to extract transistor substrate nodes allows you to perform a simple check for whether or not transistors are in properly connected (*e.g.*, grounded) wells. In a p-well CMOS process, for example, you might set the default substrate node for n-channel transistors to be some distinguished global node other than ground, *e.g.*, **NSubstrateNode!**. You could then extract the circuit, flatten it using *ext2spice* (1) (which preserves substrate nodes, unlike *ext2sim* (1) which ignores them), and look at the substrate node fields of all the n-channel transistors: if there were any whose substrate nodes weren't connected to **GND!**, then these transistors appear either outside of any explicit well (their substrate nodes will be the default of **NSubstrateNode**), or in a well that isn't tied to **GND!** with a substrate contact.

6 Extraction styles

Magic usually knows several different ways to extract a circuit from a given layout. Each of these ways is called a *style*. Different styles can be used to handle different fabrication facilities, which may differ in the parameters they have for parasitic capacitance and resistance. For a scalable technology, such as the default **scmos**, there can be a different extraction style for each scale factor. The exact number and nature of the extraction styles is described in the technology file that Magic reads when it starts. At any given time, there is one current extraction style.

To print a list of the extraction styles available, type the command

:extract style.

The **scmos** technology currently has the styles **lambda=1.5**, **lambda=1.0**, and **lambda=0.6**, though this changes over time as technology evolves. To change the extraction style to *style*, use the command

:extract style*style*

Each style has a specific scale factor between Magic units and physical units (*e.g.*, microns); you can't use a particular style with a different scale factor. To change the scalefactor, you'll have to edit the appropriate style in the **extract** section of the technology file. This process is described in "Magic Maintainer's Manual #2: The Technology File."

7 Flattening Extracted Circuits

Unfortunately, very few tools exist to take advantage of the *ext* (5) format files produced by Magic's extractor. To use these files for simulation or timing analysis, you will most likely need to convert them to a flattened format, such as *sim* (5) or *spice* (5).

There are several programs for flattening *ext* (5) files. *Ext2sim* (1) produces *sim* (5) files suitable for use with *crystal* (1), *esim* (1), or *rsim* (1). *Ext2spice* (1) is used to produce *spice* (5) files for use with the circuit-level simulator *spice* (1). Finally, *extcheck* (1) can be used to perform connectivity checking and will summarize the number of flattened nodes, transistors, capacitors, and resistors in a circuit. All of these programs make use of a library known as *extflat* (3), so the conventions for each and the checks they perform are virtually identical. The documentation for *extcheck* covers the options common to all of these programs.

To see how *ext2sim* works, load the cell **tut8n** and expand all the **tutm** subcells. Notice how the **GND!** bus is completely wired, but the **Vdd!** bus is in three disconnected pieces. Now extract everything with **:extract**, then exit Magic and run **ext2sim tut8n**. You'll see the following sort of output:

```
*** Global name Vdd!  not fully connected:
One portion contains the names:
     left/Vdd!
The other portion contains the names:
     center/Vdd!
I'm merging the two pieces into a single node, but you
should be sure eventually to connect them in the layout.

*** Global name Vdd!  not fully connected:
One portion contains the names:
     left/Vdd!
     center/Vdd!
The other portion contains the names:
     right/Vdd!
I'm merging the two pieces into a single node, but you
should be sure eventually to connect them in the layout.

Memory used:  56k
```

The warning messages are telling you that the global name **Vdd!** isn't completely wired in the layout. The flattener warns you, but goes ahead and connects the pieces together anyway to allow you to simulate the circuit as though it had been completely wired. The output of *ext2sim* will be three files: **tut8n.sim**, **tut8n.al**, and **tut8n.nodes**; see *ext2sim* (1) or *sim* (5) for more information on the contents of these files. "**Magic Tutorial #11: Using RSIM with Magic**" explains how to use the output of *ext2sim* with the switch-level simulator, *rsim* (1).

Magic Tutorial #9: Format Conversion for CIF and Calma

John Ousterhout

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

(Updated by others, too.)

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies

Commands introduced in this tutorial:

:calma, :cif

Macros introduced in this tutorial:

(None)

1 Basics

CIF (Caltech Intermediate Form) and Calma Stream Format are standard layout description languages used to transfer mask-level layouts between organizations and design tools. This tutorial describes how Magic can be used to read and write files in CIF and Stream formats. The version of CIF that Magic supports is CIF 2.0; it is the most popular layout language in the university design community. The Calma format that Magic supports is GDS II Stream format, version 3.0, corresponding to GDS II Release 5.1. This is probably the most popular layout description language for the industrial design community.

To write out a CIF file, place the cursor over a layout window and type the command

:cif

This will generate a CIF file called *name***.cif**, where *name* is the name of the root cell in the window. The CIF file will contain a description of the entire cell hierarchy in that window. If you wish to use a name different from the root cell, type the command

:cif write *file*

This will store the CIF in *file***.cif**. Start Magic up to edit **tut9a** and generate CIF for that cell. The CIF file will be in ASCII format, so you can use Unix commands like **more** and **vi** to see what it contains.

To read a CIF file into Magic, place the cursor over a layout window and type the command

:cif read *file*

This will read the file *file***.cif** (which must be in CIF format), generate Magic cells for the hierarchy described in the file, make the entire hierarchy a subcell of the edit cell, and run the design-rule checker to verify everything read from the file. Information in the top-level cell (usually just a call on the "main" cell of the layout) will be placed into the edit cell. Start Magic up afresh and read in **tut9a.cif**, which you created above. It will be easier if you always read CIF when Magic has just been started up: if some of the cells already exist, the CIF reader will not overwrite them, but will instead use numbers for cell names.

To read and write Stream-format files, use the commands **:calma read** and **:calma**, respectively. These commands have the same effect as the CIF commands, except that they operate on files with **.strm** extensions. Stream is a binary format, so you can't examine **.strm** files with a text editor.

Stream files do not identify a top-level cell, so you won't see anything on the screen after you've used the **:calma read** command. You'll have to use the **:load** command to look at the cells you read. However, if Magic was used to write the Calma file being read, the library name reported by the **:calma read** command is the same as the name of the root cell for that library.

Also, Calma format places some limitations on the names of cells: they can only contain alphanumeric characters, "$", and "_", and can be at most 32 characters long. If the name of a cell does not meet these limitations, **:calma write** converts it to a unique name of the form ___*n*, where *n* is a small integer. To avoid any possible conflicts, you should avoid using names like these for your own cells.

You shouldn't need to know much more than what's above in order to read and write CIF and Stream. The sections below describe the different styles of CIF/Calma that Magic can generate and the limitations of the CIF/Calma facilities (you may have noticed that when you wrote and read CIF above you didn't quite get back what you started with; Section 3 describes the differences that can occur). Although the discussion mentions only CIF, the same features and problems apply to Calma.

2 Styles

Magic usually knows several different ways to generate CIF/Calma from a given layout. Each of these ways is called a *style*. Different styles can be used to handle different fabrication facilities, which may differ in the names they use for layers or in the exact mask set required for fabrication. Different styles can be also used to write out CIF/Calma with slightly different feature sizes or design rules. CIF/Calma styles are described in the technology file that Magic reads when it starts up; the exact number and nature of the styles is determined by whoever wrote your technology file. There are separate styles for reading and writing CIF/Calma; at any given time, there is one current input style and one current output style.

The standard SCMOS technology file provides an example of how different styles can be used. Start up Magic with the SCMOS technology (**magic -Tscmos**). Then type the commands

:cif ostyle
:cif istyle

The first command will print out a list of all the styles in which Magic can write CIF/Calma (in this technology) and the second command prints out the styles in which Magic can read CIF/Calma. You use the **:cif** command to change the current styles, but the styles are used for both CIF and Calma format conversion. The SCMOS technology file provides several output styles. The initial (default) style for writing CIF is **lambda=1.0(gen)**. This style generates mask layers for the MOSIS scalable CMOS process, where each Magic unit corresponds to 1 micron and both well polarities are generated. See the technology manual for more information on the various styles that are available. You can change the output style with the command

:cif ostyle *newStyle*

where *newStyle* is the new style you'd like to use for output. After this command, any future CIF or Calma files will be generated with the new style. The **:cif istyle** command can be used in the same way to see the available styles for reading CIF and to change the current style.

Each style has a specific scalefactor; you can't use a particular style with a different scalefactor. To change the scalefactor, you'll have to edit the appropriate style in the **cifinput** or **cifoutput** section of the technology file. This process is described in "Magic Maintainer's Manual #2: The Technology File."

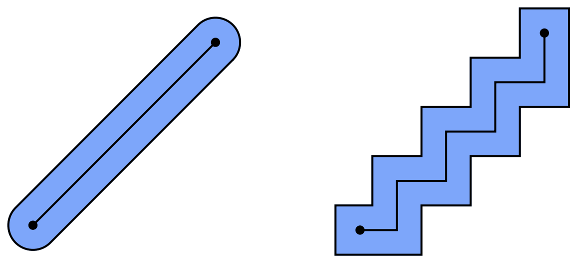
3 Rounding

The units used for coordinates in Magic are generally different from those in CIF files. In Magic, most technology files use lambda-based units, where one unit is typically half the minimum feature size. In CIF files, the units are centimicrons (hundredths of a micron). When reading CIF and Calma files, an integer scalefactor is used to convert from centimicrons to Magic units. If the CIF file contains coordinates that don't scale exactly to integer Magic units, Magic rounds the coordinates up or down to the closest integer Magic units. A CIF coordinate exactly halfway between two Magic units is rounded down. The final authority on rounding is the procedure CIFScaleCoord in the file cif/CIFreadutils.c When rounding occurs, the resulting Magic file will not match the CIF file exactly.

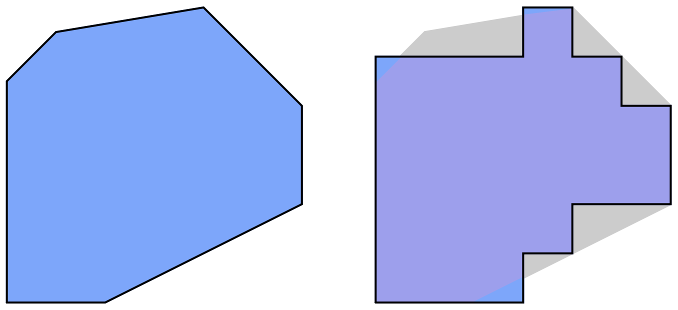
Technology files usually specify geometrical operations such as bloating, shrinking, and-ing, and or-ing to be performed on CIF geometries when they are read into Magic. These geometrical operations are all performed in the CIF coordinate system (centimicrons) so there is no rounding or loss of accuracy in the operations. Rounding occurs only AFTER the geometrical operations, at the last possible instant before entering paint into the Magic database.

4 Non-Manhattan Geometries

Magic only supports Manhattan features. When CIF or Calma files contain non-Manhattan features, they are approximated with Manhattan ones. The approximations occur for wires (if the centerline contains non-Manhattan segments) and polygons (if the outline contains non-Manhattan segments). In these cases, the non-Manhattan segments are replaced with one or more horizontal and vertical segments before the figure is processed. Conversion is done by inserting a one-unit stairstep on a 45-degree angle until a point is reached where a horizontal or vertical line can reach the segment's endpoint. Some examples are illustrated in the figure below: in each case, the figure on the left is the one specified in the CIF file, and the figure on the right is what results in Magic.



CIF Wire Resulting Magic Shape



CIF Polygon Resulting Magic Shape

The shape of the Magic stairstep depends on the order in which vertices appear in the CIF or Calma file. The stairstep is made by first incrementing or decrementing the x-coordinate, then incrementing or decrementing the y-coordinate, then x, then y, and so on. For example, in the figure above, the polygon was specified in counter-clockwise order; if it had been specified in clockwise order the result would have been slightly different.

An additional approximation occurs for wires. The CIF wire figure assumes that round caps will be generated at each end of the wire. In Magic, square caps are generated instead. The top example of the figure above illustrates this approximation.

5 Other Problems with Reading and Writing CIF

You may have noticed that when you wrote out CIF for **tut9a** and read it back in again, you didn't get back quite what you started with. Although the differences shouldn't cause any serious problems, this section describes what they are so you'll know what to expect. There are three areas where there may be discrepancies: labels, arrays, and contacts. These are illustrated in **tut9b**. Load this cell, then generate CIF, then read the CIF back in again. When the CIF is read in, you'll get a couple of warning messages because Magic won't allow the CIF to overwrite existing cells: it uses new numbered cells instead (this is why you should normally read CIF with a "clean slate"; in this case it's convenient to have both the original and reconstructed infromation present at the same time; just ignore the warnings). The information from the CIF cell appears as a subcell named **1** right on top of the old contents of **tut9b**; select **1**, move it below **tut9b**, and expand it so you can compare its contents to **tut9b**.

The first problem area is that CIF normally allows only point labels. By default, where you have line or box labels in Magic, CIF labels are generated at the center of the Magic labels. The label **in** in **tut9y** is an example of a line label that gets smashed in the CIF processing. The command

:cif arealabels yes

sets a switch telling Magic to use an extension to cif to output area-labels. This is not the default since many programs that take CIF as input do not understand this extension.

If you are reading a CIF file created by a tool other than Magic, there is an additional problems with labels. The CIF label construct ("**94** *label x y layer*") has an optional *layer* field that indicates the layer to which a label is attached. If reading a CIF file generated by Magic, this field is always present and so a label's layer is unambiguous. However, if the field is absent, Magic must decide which layer to use. It does this by looking to see what Magic layers lie beneath the label after the CIF has been read in. When there are several layers, it chooses the one appearing LATEST in the **types** section of the technology file. Usually, it's possible to ensure that the right layer is used by placing signal layers (such as metal, diffusion, and poly) later in the types section than layers such as pwell or nplus. However, sometimes Magic will still pick the wrong layer, and it will be up to you to move the label to the right layer yourself.

The second problem is with arrays. CIF has no standard array construct, so when Magic outputs arrays it does it as a collection of cell instances. When the CIF file is read back in, each array element comes back as a separate subcell. The array of **tut9y** cells is an example of this. Most designs only have a few arrays that are large enough to matter; where this is the case, you should go back after reading the CIF and replace the multiple instances with a single array. Calma format does have an array construct, so it doesn't have this problem.

The third discrepancy is that where there are large contact areas, when CIF is read and written the area of the contact may be reduced slightly. This happened to the large poly contact in **tut9b**. The shrink doesn't reduce the effective area of the contact; it just reduces the area drawn in Magic. To see what's happening here, place the box around **tut9b** and **1**, expand everything, then type

:cif see CCP

This causes feedback to be displayed showing CIF layer "CCP" (contact cut to poly). You may have to zoom in a bit to distinguish the individual via holes. Magic generates lots of small contact vias over the area of the contact, and if contacts aren't exact multiples of the hole size and spacing then extra space is left around the edges. When the CIF is read back in, this extra space isn't turned back into contact. The circuit that is read in is functionally identical to the original circuit, even though the Magic contact appears slightly smaller.

There is an additional problem with generating CIF having to do with the cell hierarchy. When Magic generates CIF, it performs geometric operations such as “grow” and “shrink”on the mask layers. Some of these operations are not guaranteed to work perfectly on hierarchical designs. Magic detects when there are problems and creates feedback areas to mark the trouble spots. When you write CIF, Magic will warn you that there were troubles. These should almost never happen if you generate CIF from designs that don't have any design-rule errors. If they do occur, you can get around them by writing cif with the following command

:cif flat *fileName*

This command creates an internal version of the design with hierarchy removed, before outputing CIF as in **cif write**. An alternative approach that does not require flattening is to modify the technology file in use. Read “Magic Maintainers Manual #2: The Technology File”, if you want to try this approach.

Magic Tutorial #10: The Interactive Router

Michael Arnold

O Division
Lawrence Livermore National Laboratory
Livermore, CA 94550

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies

Commands introduced in this tutorial:

:iroute

Macros introduced in this tutorial:

ˆR, ˆN

1 Introduction

The Magic interactive router, *Irouter*, provides an interactive interface to Magic's internal maze router. It is intended as an aid to manual routing. Routing is done one connection at a time, the user specifying a starting point and destination areas prior to each connection. The user determines the order in which signals are routed and how multi-point nets are decomposed into point-to-area connections. In addition parameters and special Magic *hint* layers permit the user to control the nature of the routes. Typically the user determines the overall path of a connection, and leaves the details of satisfying the design-rules, and detouring around or over minor obstacles, to the router.

The interactive router is not designed for fully automatic routing: interactions between nets are not considered, and net decomposition is not automatic. Thus netlists are generally not suitable input for the Irouter. However it can be convenient to obtain endpoint information from netlists. The *Net2ir* program uses netlist information to generate commands to the Irouter with appropriate endpoints for specified signals. Typically a user might setup parameters and hints to river-route a set of connections, and then generate Irouter commands with the appropriate endpoints via Net2ir. For details on Net2ir see the manual page *net2ir(1)*.

This tutorial provides detailed information on the use of the Irouter. On-line help, Irouter subcommands, Irouter parameters, and hint-layers are explained.

2 Getting Started—'Cntl-R', 'Cntl-N', ':iroute' and ':iroute help'

To make a connection with the Irouter, place the cursor over one end of the desired connection (the *start-point*) and the box at the other end (the *destination-area*). Then type

Cntl-R

Note that the box must be big enough to allow the route to terminate entirely within it. A design-rule correct connection between the cursor and the box should appear. The macro

Cntl-R

and the long commands

:iroute
:iroute route

are all equivalent. They invoke the Irouter to connect the cursor with the interior of the box. Note that the last connection is always left selected. This allows further terminals to be connected to the route with the second Irouter macro, **Cntl-N**. Try typing

Cntl-N

A connection between the cursor and the previous route should appear. In general **Cntl-N** routes from the cursor to the selection.

There are a number of commands to set parameters and otherwise interact with the Irouter. These commands have the general form

:iroute*subcommand* [*arguments*]

For a list of subcommands and a short description of each, type

:iroute help

Usage information on a subcommand can be obtained by typing

:iroute help [*subcommand*]

As with Magic in general, unique abbreviations of subcommands and most of their arguments are permitted. Case is generally ignored.

3 :Undo and Cntl-C

As with other Magic commands, the results of **:iroute** can be undone with **:undo**, and if the Irouter is taking too long it can be interrupted with **Cntl-C**. This makes it easy to refine the results of the Irouter by trial and error. If you don't like the results of a route, undo it, tweak the Irouter parameters or hints you are using and try again. If the Irouter is taking too long, you can very likely speed things up by interrupting it, resetting performance related parameters, and trying again. The details of parameters and hints are described later in this document.

4 More about Making Connections—':iroute route'

Start points for routes can be specified via the cursor, labels, or coordinates. Destination areas can be specified via the box, labels, coordinates or the selection. In addition start and destination layers can be specified explicitly. For the syntax of all these options type

:iroute help route

When a start point lies on top of existing geometry it is assumed that a connection to that material is desired. If this is not the case, the desired starting layer must be explicitly specified. When routing to the selection it is assumed that connection to the selected material is desired. By default, routes to the box may terminate on any active route layer. If you are having trouble connecting to a large region, it may be because the connection point or area is too far in the interior of the region. Try moving it toward the edge. (Alternately see the discussion of the *penetration* parameter in the wizard section below.)

5 Hints

Magic has three built-in layers for graphical control of the Irouter, **fence** (**f**), **magnet** (**mag**), and **rotate** (**r**). These layers can be painted and erased just like other Magic layers. The effect each has on the Irouter is described below.

5.1 The Fence Layer

The Irouter won't cross fence boundaries. Thus the fence layer is useful both for carving out routing-regions and for blocking routing in given areas. It is frequently useful to indicate the broad path of one or a series of routes with fence. In addition to guiding the route, the use of fences can greatly speed up the router by limiting the search.

5.2 The Magnet Layer

Magnets attract the route. They can be used to pull routes in a given direction, e.g., towards one edge of a channel. Over use of magnets can make routing slow. In particular magnets that are long and far away from the actual route can cause performance problems. (If you are having problems with magnets and performance, see also the discussion of the *penalty* parameter in the wizard section below.)

5.3 The Rotate Layer

The Irouter associates different weights with horizontal and vertical routes (see the layer-parameter section below). This is so that a preferred routing direction can be established for each layer. When two good route-layers are available (as in a two-layer-metal process) interference between routes can be minimized by assigning opposite preferred directions to the layers.

The rotate layer locally inverts the preferred directions. An example use of the rotate layer might involve an **L**-shaped bus. The natural preferred directions on one leg of the **L** are the opposite from the other, and thus one leg needs to be marked with the rotate layer.

6 Subcells

As with painting and other operations in Magic, the Irouter's output is written to the cell being edited. What the router sees, that is which features act as obstacles, is determined by the window the route is issued to (or other designated reference window - see the wizard section.) The contents of subcells expanded in the route window are visible to the Irouter, but it only sees the bounding boxes of unexpanded subcells. These bounding boxes appear on a special **SUBCELL** pseudo-layer. The spacing parameters to the **SUBCELL** layer determine exactly how the Irouter treats unexpanded subcells. (See the section on spacing parameters below.) By default, the spacings to the **SUBCELL** layer are large enough to guarantee that no design-rules will be violated, regardless of the contents of unexpanded subcells. Routes can be terminated at unexpanded subcells in the same fashion that connections to other pre-existing features are made.

7 Layer Parameters—':iroute layers'

Route-layers, specified in the **mzrouter** section of the technology file, are the layers potentially available to the Irouter for routing. The **layer** subcommand gives access to parameters associated with these route-layers. Many of the parameters are weights for factors in the Irouter cost-function. The Irouter strives for the cheapest possible route. Thus the balance between the factors in the cost-function determines the character of the routes: which layers are used in which directions, and the number of contacts and jogs can be controlled in this way. But be careful! Changes in these parameters can also profoundly influence performance. Other parameters determine which of the route-layers are actually available for routing and the width of routes on each layer. It is a good idea to inactivate route-layers not being used anyway, as this speeds up routing.

The layers subcommand takes a variable number of arguments.

:iroute layers

prints a table with one row for each route-layer giving all parameter values.

:iroute layers*type*

prints all parameters associated with route-layer *type*.

:iroute layers*type parameter*

prints the value of *parameter* for layer *type*. If *type* is '*', the value of *parameter* is printed for all layers.

:iroute layers *type parameter value*

sets *parameter* to *value* on layer *type*. If *type* is '*', *parameter* is set to *value* on all layers.

:iroute layers *type* * *value1 value2 . . . valuen*

sets a row in the parameter table.

:iroute layers ******parameter value1 . . . valuen*

sets a column in the table.
There are six layer parameters.

- **active**
  Takes the value of **YES** (the default) or **NO**. Only active layers are used by the Irouter.
- **width**
  Width of routing created by the Irouter on the given layer. The default is the minimum width permitted by the design rules.
- **hcost**
  Cost per unit-length for horizontal segments on this layer.
- **vcost**
  Cost per unit-length for vertical segments.
- **jogcost**
  Cost per jog (transition from horizontal to vertical segment).
- **hintcost**
  Cost per unit-area between actual route and magnet segment.

8 Contact Parameters—':iroute contacts'

The **contacts** subcommand gives access to a table of parameters for contact-types used in routing, one row of parameters per type. The syntax is identical to that of the **layers** subcommand described above, and parameters are printed and set in the same way.
There are three contact-parameters.

- **active**
  Takes the value of **YES** (the default) or **NO**. Only active contact types are used by the Irouter.
- **width**
  Diameter of contacts of this type created by the Irouter. The default is the minimum width permitted by the design-rules.
- **cost**
  Cost per contact charged by the Irouter cost-function.

9 Spacing Parameters—‘:iroute spacing’

The spacing parameters specify minimum spacings between the route-types (route-layers and route-contacts) and arbitrary Magic types. These spacings are the design-rules used internally by the Irouter during routing. Default values are derived from the **drc** section of the technology file. These values can be overridden in the **mzrouter** section of the technology file. (See the *Magic Maintainers Manual on Technology Files* for details.) Spacings can be examined and changed at any time with the **spacing** subcommand. Spacing values can be **nil**, **0**, or positive integers. A value of **nil** means there is no spacing constraint between the route-layer and the given type. A value of **0** means the route-layer may not overlap the given type. If a positive value is specified, the Irouter will maintain the given spacing between new routing on the specified route-layer and pre-existing features of the specified type (except when connecting to the type at an end-point of the new route).

The **spacing** subcommand takes several forms.

:iroute spacing

prints spacings for all route-types. (Nil spacings are omitted.)

:iroute spacing *route-type*

prints spacings for *route-type*. (Nil spacings are omitted.)

:iroute spacing *route-type type*

prints the spacing between *route-type* and *type*.

:iroute spacing *route-type type value*

sets the spacing between *route-type* and *type* to *value*.

The spacings associated with each route-type are the ones that are observed when the Irouter places that route-type. To change the spacing between two route-types, two spacing parameters must be changed: the spacing to the first type when routing on the second, and the spacing to the second type when routing on the first.

Spacings to the **SUBCELL** pseudo-type give the minimum spacing between a route-type and unexpanded subcells. The **SUBCELL** spacing for a given route-layer defaults to the maximum spacing to the route-layer required by the design-rules (in the **drc** section of the technology file). This ensures that no design-rules will be violated regardless of the contents of the subcell. If subcell designs are constrained in a fashion that permits closer spacings to some layers, the **SUBCELL** spacings can be changed to take advantage of this.

10 Search Parameters—‘:search’

The Mzrouter search is windowed. Early in the search only partial paths near the start point are considered; as the search progresses the window is moved towards the goal. This prevents combinatorial explosion during the search, but still permits the exploration of alternatives at all stages. The **search** subcommand permits access to two parameters controlling the windowed search, **rate**, and **width**. The **rate** parameter determines how fast the window is shifted towards the goal, and the **width** parameter gives the width of the window. The units are comparable with those used in the cost parameters. If the router is taking too long to complete, try increasing **rate**. If the router is choosing poor routes, try decreasing **rate**. The window width should probably be at least twice the rate.

The subcommand has this form:

:iroute search [*parameter*] [*value*]

If *value* is omitted, the current value is printed, if *parameter* is omitted as well, both parameter values are printed.

11 Messages—':iroute verbosity'

The number of messages printed by the Irouter is controlled by

:iroute verbosity*value*

If verbosity is set to **0**, only errors and warnings are printed. A value of **1** (the default) results in short messages. A value of **2** causes statistics to be printed.

12 Version—':iroute version'

The subcommand

:iroute version

prints the Irouter version in use.

13 Saving and Restoring Parameters—':iroute save'

The command

:iroute save *file***.ir**

saves away the current settings of all the Irouter parameters in file *file***.ir**. Parameters can be reset to these values at any time with the command

:source *file***.ir**

This feature can be used to setup parameter-sets appropriate to different routing contexts. Note that the extension **.ir** is recommended for Irouter parameter-files.

14 Wizard Parameters—':iroute wizard'

Miscellaneous parameters that are probably not of interest to the casual user are accessed via the **wizard** subcommand. The parameters are as follows:

- **bloom** Takes on a non-negative integer value. This controls the amount of compulsory searching from a focus, before the next focus is picked based on the cost-function and window position. In practice **1** (the default value) seems to be the best value. This parameter may be removed in the future.

- **boundsIncrement** Takes on the value **AUTOMATIC** or a positive integer. Determines in what size chunks the layout is preprocessed for routing. This preprocessing (blockage generation) takes a significant fraction of the routing time, thus performance may well be improved by experimenting with this parameter.

- **estimate** Takes on a boolean value. If **ON** (the default) an estimation plane is generated prior to each route that permits cost-to-completion estimates to factor in subcells and fence regions. This can be very important to efficient routing. Its rarely useful to turn estimation off.

- **expandDests** Takes on a boolean value. If **ON** (not the default) destination areas are expanded to include all of any nodes they overlap. This is particularly useful if the Irouter is being invoked from a script, since it is difficult to determine optimal destination areas automatically.

- **penalty** Takes on a rational value (default is 1024.0). It is not strictly true that the router searches only within its window. Paths behind the window are also considered, but with cost penalized by the product of their distance to the window and the penalty factor. It was originally thought that small penalties might be desirable, but experience, so far, has shown that large penalties work better. In particular it is important that the ratio between the actual cost of a route and the initial estimate is less than the value of **penalty**, otherwise the search can explode (take practically forever). If you suspect this is happening, you can set **verbosity** to **2** to check, or just increase the value of **penalty**. In summary it appears that the value of penalty doesn't matter much as long as it is large (but not so large as to cause overflows). It will probably be removed in the future.

- **penetration** This parameter takes the value **AUTOMATIC** or a positive integer. It determines how far into a blocked area the router will penetrate to make a connection. Note however the router will in no case violate spacing constraints to nodes not involved in the route.

- **window** This parameter takes the value **COMMAND** (the default) or a window id (small integers). It determines the reference window for routes. The router sees the world as it appears in the reference window, e.g., it sees the contents of subcells expanded in the reference window. If **window** is set to **COMMAND** the reference window is the one that contained the cursor when the route was invoked. To set the reference window to a fixed window, place the cursor in that window and type:

```
:iroute wizard window .
```

References

[1] M.H. Arnold and W.S. Scott, "An Interactive Maze Router with Hints", *Proceedings of the 25th Design Automation Conference*, June 1988, pp. 672–676.

Magic Tutorial #11: Using IRSIM and RSIM with Magic

Michael Chow
Mark Horowitz

Computer Systems Laboratory
Center for Integrated Systems
Stanford University
Stanford, CA 94305

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies
Magic Tutorial #8: Circuit Extraction

Commands introduced in this tutorial:

:getnode, :rsim, :simcmd, :startrsim

Macros introduced in this tutorial:

(None)

1 Introduction

This tutorial explains how to use Magic's interface to the switch-level circuit simulators, RSIM and IRSIM. The interface is the same for both these simulators and, except where noted, RSIM refers to IRSIM as well. This interface eliminates the tedium of mapping node names to objects in the layout and typing node names as RSIM input. It allows the user to select nodes using the mouse and apply RSIM commands to them or to display the node values determined by RSIM in the layout itself. You should already be familiar with using both RSIM and Magic's circuit extractor. Section 2 describes how to prepare the files necessary to simulate a circuit. Section 3 describes how to run RSIM interactively under Magic. Section 4 explains how to determine the node names that RSIM uses. Lastly, section 5 explains how to use the RSIM tool in Magic to simulate a circuit.

2 Preparations for Simulation

Magic uses the RSIM input file when it simulates the circuit. Before proceeding any further, make sure you have the correct versions of the programs **ext2sim** and **rsim** installed on your system. Important changes have been made to these programs to support simulation within Magic. To try out this tool on an example, copy all the **tut11*x*** cells to your current directory with the following command:

```
cp ˜cad/lib/magic/tutorial/tut11* .
```

The **tut11a** cell is a simple 4-bit counter using the Magic scmos technology file. Start Magic on the cell **tut11a**, and extract the entire cell using the command:

```
:extract all
```

When this command completes, several **.ext** files will be created in your current directory by the extractor. The next step is to flatten the hierarchy into a single representation. Return to the Unix c-shell by quitting Magic.

The program **ext2sim** is used to flatten the hierarchy. Run this program from the C-shell by typing:

```
ext2sim -L -R -c 20 tut11a
```

This program will create the file **tut11a.sim** in your current directory.

If you are running IRSIM, the **tut11a.sim** can be used directly as input to the simulator and you should skip the next step. Instead, if you will be using RSIM, the last step is to create the binary representation of the flattened hierarchy by using the program **presim**. To do this, type:

```
presim tut11a.sim tut11a.rsm ˜cad/lib/scmos100.prm -nostack -nodrops
```

The third file is the parameter file used by presim for this circuit. The convention at Stanford is to use the suffix *.rsm* when naming the RSIM input file. The file **tut11a.rsm** can also be used as input for running RSIM alone.

3 Using RSIM

Re-run Magic again to edit the cell **tut11a**. We'll first learn how to run RSIM in interactive mode under Magic. To simulate the circuit of tut11a, using IRSIM type the command:

```
:rsim scmos100.prm tut11a.sim
```

To simulate the circuit of tut11a, using RSIM type the command:

```
:rsim tut11a.rsm
```

You should see the RSIM header displayed, followed by the standard RSIM prompt (**rsim**> or **irsim**>, depending on the simulator) in place of the usual Magic prompt; this means keyboard input is now directed to RSIM. This mode is very similar to running RSIM alone; one difference is that the user can escape RSIM and then return to Magic. Also, the mouse has no effect when RSIM is run interactively under Magic.

Only one instance of RSIM may be running at any time under Magic. The simulation running need not correspond to the Magic layout; however, as we shall see later, they must correspond for the RSIM tool to work. All commands typed to the RSIM prompt should be RSIM commands. We'll first run RSIM, then escape to Magic, and then return back to RSIM. Type the RSIM command

@ tut11a.cmd

to initialize the simulation. (Note there is a " " after the @.) Now type **c** to clock the circuit. You should see some information about some nodes displayed, followed by the time. Set two of the nodes to a logic "1" by typing **h RESET_B hold**. Step the clock again by typing **c**, and RSIM should show that these two nodes now have the value "1".

You can return to Magic without quitting RSIM and then later return to RSIM in the same state in which it was left. Escape to Magic by typing:

.

(a single period) to the RSIM prompt. Next, type a few Magic commands to show you're really back in Magic (signified by the Magic prompt).

You can return to RSIM by typing the Magic command **rsim** without any arguments. Type:

:rsim

The RSIM prompt will be displayed again, and you are now back in RSIM in the state you left it in. Experiment with RSIM by typing some commands. To quit RSIM and return to Magic, type:

q

in response to the RSIM prompt. You'll know you're back in Magic when the Magic prompt is redisplayed. If you should interrupt RSIM (typing a control-C), you'll probably kill it and then have to restart it. RSIM running standalone will also be killed if you interrupt it. If you interrupt IRSIM (typing a control-C), the simulator will abort whatever it's doing (a long simulation run, for example) and return to the command interpreter by prompting again with **irsim**>.

4 Node Names

It's easy to determine node names under Magic. First, locate the red square region in the middle right side of the circuit. Move the cursor over this region and select it by typing **s**. To find out the name for this node, type:

:getnode

Magic should print that the node name is *RESET_B*. The command **getnode** prints the names of all nodes in the current selection. Move the cursor over the square blue region in the upper right corner and add this node to the current selection by typing **S**. Type **:getnode** again, and Magic should print the names of two nodes; the blue node is named *hold*. You can also print aliases for the selected nodes. Turn on name-aliasing by typing:

:getnode alias on

Select the red node again, and type **:getnode**. Several names will be printed; the last name printed is the one RSIM uses, so you should use this name for RSIM. Note that **getnode** is not guaranteed to print all aliases for a node. Only those alises generated when the RSIM node name is computed are printed. However, most of the alaiases will usually be printed. Printing aliases is also useful to monitor the name search, since **getnode** can take several seconds on large nodes. Turn off aliasing by typing:

:getnode alias off

getnode works by extracting a single node. Consequently, it can take a long time to compute the name for large nodes, such as *Vdd* or *GND*. Select the horizontal blue strip on top of the circuit and run **:getnode** on this. You'll find that this will take about six seconds for **getnode** to figure out that this is *Vdd*. You can interrupt **getnode** by typing **ˆC** (control-C), and **getnode** will return the "best" name found so far. There is no way to tell if this is an alias or the name RSIM expects unless **getnode** is allowed to complete. To prevent these long name searches, you can tell **getnode** to quit its search when certain names are encountered. Type:

:getnode abort Vdd

Select the blue strip on top of the circuit and type **:getnode**. You'll notice that the name was found very quickly this time, and **getnode** tells you it aborted the search of *Vdd*. The name returned may be an alias instead of the the one RSIM expects. In this example, the abort option to **getnode** will abort the name search on any name found where the last component of the node name is *Vdd*. That is, **getnode** will stop if a name such as "miasma/crock/*Vdd*" or "hooha/*Vdd*" is found.

You can abort the search on more than one name; now type **:getnode abort GND**. Select the bottom horizontal blue strip in the layout, and type **:getnode**. The search will end almost immediately, since this node is *GND*. **getnode** will now abort any node name search when either *Vdd* or *GND* is found. The search can be aborted on any name; just supply the name as an argument to **getnode abort**. Remember that only the last part of the name counts when aborting the name search. To cancel all name aborts and resume normal name searches, type:

:getnode abort

getnode will no longer abort the search on any names, and it will churn away unless interrupted by the user.

5 RSIM Tool

You can also use the mouse to help you run RSIM under Magic. Instead of typing node names, you can just select nodes with the mouse, tell RSIM what to do with these nodes, and let Magic do the rest. Change tools by typing:

:tool rsim

or hit the space bar until the cursor changes to a pointing hand. The RSIM tool is active when the cursor is this hand. The left and right mouse buttons have the same have the same function as the box tool. You use these buttons along with the select command to select the nodes. The middle button is different from the box tool. Clicking the middle button will cause all nodes in the selection to have their logical values displayed in the layout and printed in the text window. We need to have RSIM running in order to use this tool. Start RSIM by typing:

:startrsim tut11a.rsm

The **.rsm** file you simulate must correspond to the root cell of the layout. If not, Magic will generate node names that RSIM will not understand and things won't work properly. If any paint is changed in the circuit, the circuit must be re-extracted and a new **.rsm** file must be created to reflect the changes in the circuit.

Magic will print the RSIM header, but you return to Magic instead of remaining in RSIM. This is an alternate way of starting up RSIM, and it is equivalent to the command **rsim tut11a.rsm** and typing a period (**.**) to the RSIM prompt, escaping to Magic. We need to initialize RSIM, so get to RSIM by typing **:rsim** and you'll see the RSIM prompt again. As before, type **@ tut11a.cmd** to the RSIM prompt to initialize everything. Type a period (**.**) to return to Magic. We are now ready to use the RSIM tool.

As mentioned earlier, **tut11a** is a 4-bit counter. We'll reset the counter and then step it using the RSIM tool. Locate the square blue area on the top right corner of the circuit. Place the cursor over this region and select it. Now click the middle button, and the RSIM value for this node will be printed in both the text window and in the layout. Magic/RSIM will report that the node is named *hold* and that its current value is *X*. You may not be able to see the node value in the layout if you are zoomed out too far. Zoom in closer about this node if necessary. Try selecting other nodes, singly or in groups and click the middle button to display their values. This is an easy way to probe nodes when debugging a circuit.

Select *hold* again (the blue square). This node must be a "1" before resetting the circuit. Make sure this is the only node in the current selection. Type:

:simcmd h

to set it to a "1". Step the clock by typing:

:simcmd c

Click the middle button and you will see that the node has been set to a “1.” The Magic command **simcmd** will take the selected nodes and use them as RSIM input. These uses of **simcmd** are like typing the RSIM commands *h hold* followed by *c*. The arguments given to **simcmd** are normal RSIM commands, and **simcmd** will apply the specified RSIM command to each node in the current selection. Try RSIM commands on this node (such as *?* or *d*) by using the command as an argument to **simcmd**.

You can enter RSIM interactively at any time by simply typing **:rsim**. To continue using the RSIM tool, escape to Magic by typing a period (**.**) to the RSIM prompt.

The node *RESET_B* must be set to a “0”. This node is the red square area at the middle right of the circuit. Place the cursor over this node and select it. Type the Magic commands **:simcmd l** followed by **:simcmd c** to set the selected node to a “0”. Click the middle mouse button to check that this node is now “0”. Step the clock once more to ensure the counter is reset. Do this using the **:simcmd c** command.

The outputs of this counter are the four vertical purple strips at the bottom of the circuit. Zoom in if necessary, select each of these nodes, and click the middle button to check that all are “0”. Each of these four nodes is labeled *bit_x*. If they are all not “0”, check the circuit to make sure *hold=1* and *RESET_B=0*. Assuming these nodes are at their correct value, you can now simulate the counter. Set *RESET_B* to a “1” by selecting it (the red square) and then typing **:simcmd h**. Step the clock by typing **:simcmd c**. Using the same procedure, set the node *hold* (the blue square) to a “0”.

We’ll watch the output bits of this counter as it runs. Place the box around all four outputs (purple strips at the bottom) and zoom in so their labels are visible. Select one of the outputs by placing the cursor over it and typing **s**. Add the other three outputs to the selection by placing the cursor over each and typing **S**. These four nodes should be the only ones in the selection. Click the middle mouse button to display the node values. Step the clock by typing **:simcmd c**. Click the middle button again to check the nodes. Repeat stepping the clock and displaying the outputs several times, and you’ll see the outputs sequence as a counter. If you also follow the text on the screen, you’ll also see that the outputs are also being watched.

You may have noticed that the results are printed very quickly if the middle button is clicked a second time without changing the selection. This is because the node names do not have to be recomputed if the selection remains unchanged. Thus, you can increase the performance of this tool by minimizing selection changes. This can be accomplished by adding other nodes to the current selection that you are intending to check.

To erase all the RSIM value labels from the layout, clear the selection by typing:

```
:select clear
```

and then click the middle mouse button. The RSIM labels do not affect the cell modified flag, nor will they be written in the **.mag** file. When you’re finished using RSIM, resume RSIM by typing **:rsim** and then quit it by typing a **q** to the RSIM prompt. Quitting Magic before quitting RSIM will also quit RSIM.

We’ve used a few macros to lessen the typing necessary for the RSIM tool. The ones commonly used are:

```
:macro h “simcmd h”
:macro l “simcmd l”
```

:macro k “simcmd c”

Magic Tutorial #S-1: The scheme command-line interpreter

Rajit Manohar

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Read reference [1]

Commands introduced in this tutorial:

:scm-echo-result, :eval, lots of scheme functions

Macros introduced in this tutorial:

(None)

1 Introduction

Magic's original command-line interpreter has some limitations. Some of these include the absence of definitions with arguments, block structure, the ability to define complex functions. We describe an extension which is almost completely backward compatible with the existing magic command-line syntax, but permits the use of Scheme on the command-line.

2 Backward compatibility

To permit existing magic source files to work within the scheme interpreter, we have had to sacrifice one feature of the magic command-line syntax. Single quotes can only be used to quote a single character. The reason for this limitation is that *unmatched* quotes are used by scheme to stop evaluation of the next input symbol.

Parentheses are used by the scheme interpreter. If you use parentheses outside single or double quotes in your magic source files, you might find that the source files don't work properly. To circumvent this problem, simply put your parentheses in double quotes. You can also use backslashes to quote parentheses as in:

```
:macro \( "echo hello"
```

Another thing you may notice is that floating-point numbers are parsed as such, and therefore a command such as

```
:echo 5.3
```

would display the string **5.300000**. If you really want the string **5.3**, use:

```
:echo "5.3"
```

If this difference is undesirable, the scheme interpreter can be turned off at compile-time. Talk to your local magic maintainer if you want this done. We feel that the minor trouble taken in modifying existing magic source files will be outweighed by the advantage of using a more powerful layout language.

3 The scheme interpreter

The interpreter supports a subset of the scheme language. The features of scheme that are missing include character types, vector types, file input/output, complex numbers, the distinction between exact and inexact arithmetic, quasi-quotations, and continuations.

3.1 Command-line interaction

When interacting with the command-line of magic, the interpreter implicitly parenthesizes its input. For example, the command

```
:paint poly
```

would be interpreted as the scheme expression

```
(paint poly)
```

This has exactly the same effect as the original expression, because all existing magic command-line functions are also scheme functions. Since the valid magic commands vary from window to window, the return value of the function is a boolean that indicates whether the command was valid for the current window.

The boolean **scm-echo-result** controls whether or not the result of the evaluation is displayed. If the variable does not exist, or the variable is not boolean-valued, the result of evaluation is not echoed. Since the input is implicitly parenthesized, typing in

:scm-echo-result

would not display the value of the variable, since it would be evaluated as:

(scm-echo-result)

To display the value of a variable, use the built-in procedure **eval** as follows:

:eval scm-echo-result

This would result in the expression:

(eval scm-echo-result)

which would have the desired effect (note that for this to actually display anything, the value of **scm-echo-result** must be **#t**, and so examining its value is really a futile exercise—which is why it is an example, of course!).

3.2 Types of arguments

Since scheme expressions are typed, we may need to examine the type of a particular expression. The following functions return booleans, and can be used to determine the type of an object.

#t and **#f** are constants representing the booleans true and false. A standard scheme convention is to name functions that return booleans with a name ending with "?". The built-in functions conform to this convention.

The expression *expr* is evaluated, and the type of the result of evaluation is checked.

(boolean? *expr***)**	**#t** if *expr* is a boolean
(symbol? *expr***)**	**#t** if *expr* is a symbol
(list? *expr***)**	**#t** if *expr* is a list
(pair? *expr***)**	**#t** if *expr* is a pair
(number? *expr***)**	**#t** if *expr* is a number
(string? *expr***)**	**#t** if *expr* is a string
(procedure? *expr***)**	**#t** if *expr* is a procedure

For example,

(boolean? #t)=> *#t*
(number? #t)=> *#f*

3.3 Lists and pairs

A pair is a record structure with two fields, called the car and cdr fields (for historical reasons). Pairs are used primarily to represent lists. A list can be defined recursively as either the empty list, or a pair whose cdr field is a list. The following functions are used to extract these fields and to construct new pairs and lists.

(car *pair***)**	the car field of *pair*
(cdr *pair***)**	the cdr field *pair*
(cons *obj1 obj2***)**	a new pair whose car field is *obj1* and cdr field is *obj2*
(list *arg1 . . .* **)**	a new list consisting of its arguments
(null? *list***)**	**#t** if *list* is the empty list
(length *list***)**	the number of elements in *list*

For example,

(car '(a b c))=> *a*
(cdr '(a b c))=> *(b c)*
(cons 'a '(b c))=> *(a b c)*
(list 'a 'b 'c)=> *(a b c)*
(null? '(a b))=> *#f*
(null? ())=> *#t*

The car field and cdr field of a pair can be set using the following two functions.

(set-car! *pair obj***)**	sets the car field of *pair* to *obj*. It returns the new pair
(set-cdr! *pair obj***)**	sets the cdr field of *pair* to *obj*. It returns the new pair

These two functions have *side-effects*, another feature that distinguishes scheme from pure lisp. Another naming convention followed is that functions that have side-effects end in "!".

Try the following sequence in magic:

(define x '(a b))=> *(a b)*
(set-car! x 'c)=> *(c b)*
(set-cdr! x '(q))=> *(c q)*
(set-cdr! x 'q)=> *(c . q)*

After the last statement, the value of x is no longer a list but a pair.

3.4 Arithmetic

The interpreter supports both floating-point and integer arithmetic. The basic arithmetic functions are supported.

(+ *num1 num2***)**	the sum *num1+num2*
(- *num1 num2***)**	the difference *num1-num2*
(* *num1 num2***)**	the product *num1*num2*
(/ *num1 num2***)**	the quotient *num1/num2*
(truncate *num***)**	the integer part of *num*

The division operator checks for division by zero, and promotes integers to floating-point if deemed necessary. Floating-point numbers can be converted into integers by truncation. The range of a number can be checked using the following predicates:

(zero? *num***)**	**#t** if *num* is zero
(positive? *num***)**	**#t** if *num* is positive
(negative? *num***)**	**#t** if *num* is negative

3.5 Strings

The interpreter supports string manipulation. String manipulation can be useful for interaction with the user as well as constructing names for labels.

(string-append *str1 str2***)**	the string formed by concatenating *str1* and *str2*
(string-length *str***)**	the length of string *str*
(string-compare *str1 str2***)**	a positive, zero, or negative number depending on whether *str1* is lexicographically greater, equal to, or less than *str2*
(string-ref *str int***)**	the numerical value of the character stored at position *int* in *str* (The first character is at position 0.)
(string-set! *str int1 int2***)**	sets character in string *str* at position *int1* to *int2*
(substring *str int1 int2***)**	returns substring of *str* from position *int1* (inclusive) to *int2* (exclusive)

Strings can be used to convert to and from various types.

(number->string *num***)**	the string corresponding to the representation of *num*
(string->number *str***)**	the number corresponding to *str*
(string->symbol *str***)**	a symbol named *str*
(symbol− >string *sym***)**	the string corresponding to the name of *sym*

3.6 Bindings and functions

An object (more accurately, the *location* where the object is stored) can be bound to a symbol using the following two functions:

(define *sym expr***)**	bind *expr* to *sym*, creating a new symbol if necessary and return *expr*
(set! *sym expr***)**	bind *expr* to an existing symbol *sym* and return *expr*

(Note: these functions do not evaluate their first argument.)

The difference between the two is that **define** introduces a new binding, whereas **set!** modifies an existing binding. In both cases, *expr* is evaluated, and the result is bound to the symbol *sym*. The result of the evaluation is also returned.

(define x 4)=> *4*

Functions can be defined using lambda expressions. Typically a function is bound to a variable. If required, a lambda expression or built-in function can be applied to a list.

(lambda *list obj***)**	a new function

(Note: a lambda does not evaluate its arguments.)

list is a list of symbol names, and *obj* is the expression that corresponds to the body of the function. For example,

(lambda (x y z) (+ (+ x y) z))=> *#proc*

is a function that takes three arguments and returns their sum. It can be bound to a symbol using **define**.

(define sum3 (lambda (x y z) (+ (+ x y) z)))=> *#proc*

Now, we can use **sum3** like any other function.

(sum3 5 3 8)=> *16*

A function can be applied to a list using **apply**.

(apply *proc list***)**	apply *proc* to *list*

(Note: both *proc* and *list* are evaluated before application.)

list is used as the list of arguments for the function. For instance, an alternative way to sum the three numbers in the example above is:

(apply sum3 '(3 5 8))=> *16*

An alternative method for creating bindings is provided by the **let** mechanism.

(let *binding-list expr***)**	evaluate *expr* after the bindings have been performed
(let* *binding-list expr***)**	evaluate *expr* after the bindings have been performed
(letrec *binding-list expr***)**	evaluate *expr* after the bindings have been performed

The *binding-list* is a list of bindings. Each binding is a list containing a symbol and an expression. The expression is evaluated and bound to the symbol. In the case of **let**, all the expressions are evaluated before binding them to any symbol; **let***, on the other hand, evaluates an expression and binds it to the symbol before evaluating the next expression. **letrec** permits bindings to refer to each other, permitting mutually recursive function definitions. The evaluation order is defined to be from left to right in all cases. After performing the bindings, *expr* is evaluated with the new bindings in effect and the result is returned.

let bindings can be used in interesting ways. An example of their use is provided later.

Scheme is an eager language, and only a few functions do not evaluate all their arguments (definitions and conditionals). Evaluation can be controlled to some degree using the following two functions:

(quote *obj***)**	the unevaluated object *obj*
(eval *obj***)**	evaluates object *obj*

3.7 Control structures

Since scheme is a functional programming language, functions that are usually written using loops are written using recursion. Conditional constructs are used to terminate the recursion. These constructs are slightly different in that they do not evaluate all their arguments (otherwise recursive functions would not terminate!).

(if *expr arg1 arg2***)**	evaluate *expr* and evaluate one of *arg1* or *arg2*

The **if** construct evaluates its first argument (which must result in a boolean), and if the result is **#t** evaluates *arg1* and returns the result; otherwise *arg2* is evaluated and returned.

For instance, the standard factorial function might be written as:

(define fact (lambda (x) (if (positive? x) (* x (fact (- x 1))) 1)))

A more complicated form of conditional behavior is provided by **cond**.

(cond *arg1 arg2 ...***)**	generalized conditional

Each argument consists of a list which contains two expressions. The first expression is evaluated (and must evaluate to a boolean), and if it is true the second expression is evaluated and returned as the result of the entire expression. If the result was false, the next argument is examined and the above procedure is repeated. If all arguments evaluate to false, the result is undefined.

For instance if **x** was a list, the expression

(cond ((null? x) x) ((list? x) (car x)) (#t (echo "error")))

would return the empty list if **x** was the empty list and the first element from the list otherwise. When **x** is not a list, an error message is displayed. Note that **echo** is a standard magic command.

Often one needs to evaluate a number of expressions in sequence (since the language has side-effects). The **begin** construct can be used for this purpose.

(begin *arg1 arg2 . . .* **)**	sequencing construct

begin evaluates each of its arguments in sequence, and returns the result of evaluating its last argument.

3.8 Interaction with layout

All standard magic commands are also scheme functions. This permits one to write scheme functions that interact with the layout directly. Apart from the standard magic commands, the following scheme functions are provided so as to enable the user to edit layout.

(getbox)	a list containing four members (llx lly urx ury)
(getpaint *str***)**	a list containing the boxes from layer *str* under the current box that have paint in them
(getlabel *str***)**	a list containing the labels under the current box that match *str*
(magic *sym***)**	forces *sym* to be interpreted as a magic command

The pairs (llx,lly) and (urx,ury) correspond to magic coordinates for the lower left and upper right corner of the current box. **getpaint** returns a list of boxes (llx lly urx ury), and **getlabel** returns a list of tagged boxes (label llx lly urx ury) which contain the label string. **magic** can be used to force the scheme interpreter to interpret a symbol as a magic procedure. The evaluation returns the specified magic command.

3.9 Miscellaneous

Some additional functions are provided to enable the user to debug functions.

(showframe)	display the current list of bindings
(display-object *obj***)**	display the type and value of *obj*
(error *str***)**	display error message and abort evaluation
(eqv? *obj1 obj2***)**	checks if two objects are equal
(collect-garbage)	force garbage collection

The following is a complete list of the built-in scheme variables that can be used to control the interpreter.

scm-library-path	a colon-separated path string
scm-echo-result	a boolean used to determine if the result of evaluation should be displayed
scm-trace-magic	controls display of actual magic commands
scm-echo-parser-input	controls display of the string sent to the scheme parser
scm-echo-parser-output	controls display of the result of parsing
scm-stack-display-depth	controls the number of frames displayed in the stack trace output when an error occurs during evaluation
scm-gc-frequency	controls the frequency of garbage collection

3.10 Libraries

The following function loads in a file and evaluates its contents in order.

(load-scm *str***)**	reads scheme commands in from the named file
(save-scm *str obj***)**	appends *obj* to the file *str*, creating a new file if necessary

The file can be in the current directory, or in any of the locations specified by a string containing a colon-separated list of directory names stored in **scm-library-path**.

The format of these files differs from standard magic source files because the contents of a line are not implicitly parenthesized. In addition, semicolons are used as a comment character; everything following a semicolon to the end of the current line is treated as a comment.

For instance,

```
define f (lambda (x) x)
```

would define **f** to be the identity function when placed in a magic source file (so as to provide backward compatibility). The same definition would result in an error if placed in a scheme source file.

(define f (lambda (x) x))

The above expression should be used in the scheme file to achieve the same effect.

References

[1] H. Abelson and G.J. Sussman, *Structure and Interpretation of Computer Programs*.

[2] H. Abelson *et al.*, *Revised Report on the Algorithmic Language Scheme*.

Magic Tutorial #S-2: Boxes and labels

Rajit Manohar

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #S-1: The scheme command-line interpreter

Commands introduced in this tutorial:

:getbox, :box.push, :box.pop, :box.move, :label.vert, :label.horiz, :label.rename, :label.search, :label.find-next

Macros introduced in this tutorial:

(None)

1 The current box

The fundamental way scheme programs interact with magic layout is by using magic's **box** command. For instance,

```
(box 1 1 2 2)
```

changes the current box to the rectangle defined by the coordinates (1,1) and (2,2) in the current edit cell. This is the standard magic **:box** command. After moving the box to a particular position in the layout, the area can be painted, erased, selected, etc.

The scheme function **getbox** returns the current box as a list of four integers. For instance,

```
(box 1 1 2 2)
(define x (getbox))
```

will bind the list **(1 1 2 2)** to variable **x**.

2 Saving and restoring the box

If a scheme function moves the current box around, it is good practice to restore the box back to its original position. This is especially useful when writing a function that the user is likely to type on the command line.

box.push can be used to push a box onto the current stack of boxes. **box.pop** restores the box to the one on the top of the box stack. The sequence

```
(box.push (getbox))
(box 1 1 5 4)
(paint poly)
(box.pop)
```

will paint a rectangle of polysilicon from (1,1) to (5,4), restoring the original position of the box.

3 Moving the box

Magic's built-in **move** command is not entirely reliable. Sometimes move commands are ignored, with disastrous effects. (Think about what might happen if a move command was ignored in the middle of drawing a stack of twenty transistors . . .) The scheme function **box.move** moves the box relative to the current position.

```
(box.move 5 3)
```

will move the box right 5 lambda and up 3 lambda.

4 Labelling vertical and horizontal wires

Datapaths are usually designed by designing cells for a single bit of the datapath, and then arraying those cells to obtain the complete datapath. When simulating such designs, it is usually desirable to label wires in the datapath with names like "name0", "name1", up to "nameN."

There are two functions that can be used to perform such a task. The function **label.vert** returns a function that can be used as a labeller for vertically arrayed nodes. **label.horiz** returns a function that can be used as a labeller for horizontally arrayed nodes.

```
(define lbl (label.vert "name" 6)
```

The command above defines a new function **lbl** that can be used to generate labels beginning with "name0" for nodes that are vertically spaced by 6 lambda. The simplest way to use this function is to bind it to a macro as follows:

```
(macro 1 "lbl")
```

Place the box over the lowest node. Every time key "1" is pressed, a new label "nameM" is created and the box is moved up by 6 lambda. **label.horiz** can be used in a similar fashion for labelling nodes that are horizontally arrayed.

5 Finding and renaming existing labels

The label macros provide functionality to search for all labels that match a particular string. Place the box over the region of interest. Type:

```
(label.search "label")
```

To place the box over the first occurrence of the label you searched for, type:

```
(label.find-next)
```

Repeatedly executing this function causes the box to move to all the labels that match the search pattern. Typically, one would bind **label.find-next** to a macro.

The command **label.rename** can be used to rename all labels with a particular name. To use this command, place the box over the region of interest. Then type

```
(label.rename "label1" "label2")
```

All occurrences of label “label1” in the current box will be renamed to “label2”.

6 Writing these functions

The functions discussed in this tutorial are not built-in. They are user-defined functions in the default scheme file loaded in when magic starts.

As you begin to use magic with the scheme command-line interpreter, you will observe that commands for drawing paint on the screen are extremely slow. This time interval is not normally noticeable because editing is interactive. However, when one can write a scheme program to draw twenty transistors on the screen, this delay becomes noticeable. It is worthwhile to minimize the number of magic commands executed, even if this involves writing more scheme code. The **box-pop** command has been tuned a little to not execute the **box** command if the box would not move as a result.

```
(define box.list ())

(define box.move
        (lambda (dx dy)
                (let* ((x (getbox))
                       (nllx (+ dx (car x)))
                       (nlly (+ dy (cadr x)))
                       (nurx (+ dx (caddr x)))
                       (nury (+ dy (cadddr x))))
                (box nllx nlly nurx nury)
                )
        )
)
```

```
(define box.=?
        (lambda (b1 b2)
                (and (and (=? (car b1) (car b2)) (=? (cadr b1) (cadr b2)))
                      (and (=? (caddr b1) (caddr b2)) (=? (caddr b1) (caddr b2)))
                )
        )
)

(define box.push
        (lambda (pos)
                (set! box.list (cons pos box.list))
        )
)

(define box.pop
        (lambda ()
                (if (null? box.list)
                      (echo "Box list is empty")
                      (let ((x (car box.list)))
                              (begin
                              (set! box.list (cdr box.list))
                              (if (box.=? x (getbox)) #t (eval (cons 'box x)))
                              )
                      )
                )
        )
)
```

Magic Tutorial #S-3: Transistor stacks

Rajit Manohar

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #S-1: The scheme command-line interpreter

Commands introduced in this tutorial:

:stack.p, :stack.n, :stack.tallp, :stack.talln, :prs.draw, :prs.mgn, :prs.talldraw, :prs.tallmgn

Macros introduced in this tutorial:

(None)

1 Stacks

The first step in laying out a gate/operator using magic tends to involve drawing the transistor stacks without any wiring, labelling all the important nodes in the circuit. Since the extractor pretends that nodes that have the same label are electrically connected, the extracted circuit can be simulated using SPICE to obtain some indication of the power/speed of the circuit.

stack.tallp and **stack.talln** can be used to draw such transistor stacks and place contacts where required. These two functions take a transistor width and a list of strings that represent the stack as their arguments, and draw the stack vertically (gates run horizontally) at the current box. For example,

(stack.tallp 40 '(("Vdd") "a" "b" ("Inode") "d" ("out")))

draws a vertical stack of p-transistors with the diffusion being 40 lambda wide. The stack begins with a contact labelled “Vdd”, followed by two gates labelled “a” and “b”, followed by a contact labelled “Inode”, followed by a gate labelled “d”, followed by a contact labelled “out”. Contacts are indicated by placing the string for the label in parenthesis. Note the presence of the quote that prevents the interpreter from attempting to evaluate the list.

The contact width and contact-gate spacing together amount to more than the spacing between adjacent gates. Often it is desired to eliminate this extra space by jogging the poly wires so that the amount of internal diffusion capacitance is minimized. The functions **stack.p** and **stack.n** can be used to do so. Typing

(stack.tallp 40 ’(("Vdd") "a" "b" ("Inode") "d" ("out")))

will draw the same stack and jog the poly wires corresponding to the gate for “d”.

2 Production rules

The functions **prs.draw** and **prs.mgn** can be used to draw the transistor stacks that implement a production rule. For instance,

(prs.draw 40 "x & y − > z-")

will draw the stack required to implement the specified production rule with width 20. The function takes a gate width and a single production rule as its arguments. Note that the production rules must be in negation normal form, i.e., all negations must be on variables.

Contacts can be shared between operators by providing a list of production rules as input to **prs.mgn**, as follows:

(prs.mgn 40 20 "x & y − > z-" "u & v − > w-" "˜x & ˜y − > z+")

The contact to GND will be shared between the pull-down stacks for z and w.

Both production-rule drawing function ensure that the variable order in the production rule (from left to right) corresponds to the gate order in the transistor stacks (from power supply to output).

It is not always possible to directly draw a production rule in a single stack with no additional internal contacts. In this case, the function creates an internal node name of the form “_” followed by a number. You can search for these using **(label.search "_*")**, followed by **(label.find-next)**. To complete the implementation, you will have to wire up these internal contacts as well.

Both **prs.mgn** and **prs.draw** draw the transistor stacks using **stack.p** and **stack.n**. The variants **prs.tallmgn** and **prs.talldraw** use **stack.tallp** and **stack.talln** instead.

Magic Tutorial #S-4: The design rule file

Rajit Manohar

Deparment of Computer Science
California Institute of Technology
Pasadena, CA 91125

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #S-1: The scheme command-line interpreter

Commands introduced in this tutorial:

:drc.

Macros introduced in this tutorial:

(None)

1 Introduction

Magic Tcl Tutorial #1: Introduction

R. Timothy Edwards

Space Department
Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20723

This tutorial corresponds to Tcl-based Magic version 7.2

1 What is Tcl-based Magic, and Why?

In Magic version 7.0, Rajit Manohar incorporated a SCHEME interpreter into the Magic source, noting the limitation of magic to handle definitions and variables, conditionals, and block structures. By embedding an interpreter into the code, the interpreter's functions are made available on the magic command line, making magic extensible. The SCHEME interpreter and various extensions incorporated into loadable scripts are outlined in the tutorials `tutscm1.ps` through `tutscm4.ps`.

While making Magic considerably more flexible, the embedded SCHEME interpreter had some notable drawbacks. The primary one is that the SCHEME language is syntactically different from Magic's command-line syntax. Also, the interpreter is largely disconnected from the code, and does not affect or extend the graphics or handle results from magic commands.

Beginning in Magic version 7.2, Magic has been recast into a framework called *ScriptEDA*, in which existing applications become **extensions** of an interpreter rather than having an interpreter embedded in the application. The main advantage of extending over embedding is that the application becomes a module of the interpreter language, which does not preclude the use of additional, unrelated modules in the same interpretive environment. For example, in Tcl-based Magic, graphics are handled by Tk (the primary graphics package for use with Tcl), and applications such as IRSIM (the digital simulator) can be run as if they were an extension of magic itself. Commands for Tcl, Tk, IRSIM, BLT, and any other Tcl-based package can be mixed on the magic command line.

While *ScriptEDA* suggests the use of the **SWIG** package to give applications the ability to be compiled as extensions of any interpreter (Tcl, Python, SCHEME, Perl, etc.), there are specific advantages to targeting Tcl. Foremost, the syntax of Tcl is virtually 100% compatible with that of Magic. This is not coincidentally because both Magic and Tcl were written by John Ousterhout! Many ideas from the development of Magic were incorporated into the overall concept and design of the Tcl interpreter language.

Largely due to the syntactical compatibility, Tcl-based magic is completely backwardly-compatible with the non-interpreter version of magic. Either can be selected at compile-time, in addition to compiling with embedded SCHEME, which has been retained in full as an option. A few minor issues, such as the appearance of the cursor when magic is used without a Tk console window, are addressed below.

Magic extensions under Tcl are considerable, and explanations of the features have been split into several tutorial files, as listed in Table 1.

Magic Tcl Tutorial #1:	Introduction
Magic Tcl Tutorial #2:	The GUI Wrapper
Magic Tcl Tutorial #3:	Extraction and Netlisting
Magic Tcl Tutorial #4:	Simulation with IRSIM
Magic Tcl Tutorial #5:	Writing Tcl Scripts for Magic

Table 1: The Magic Tcl tutorials and other documentation.

2 Features of Tcl-based Magic

In summary, the features of Tcl-based Magic (corresponding to Magic version 7.2, revision 31) are as follows:

1. The command name **magic** itself is a script, not a compiled executable. The script launches Tcl and gives it the name of a Tcl script to evaluate (`magic.tcl`). The Tcl script loads Magic as an extension of Tcl and performs all other operations necessary to start up the application.

2. Command-line arguments passed to Magic have been changed. Some older, unused arguments have been removed. Several arguments have been added, as follows:

   (a) `-noconsole` Normally, under Tcl/Tk, Magic starts by launching a Tk-based console window (`tkcon.tcl`) which is the window that accepts magic commands. Previous versions of Magic accepted commands from the calling terminal. The former style of running commands from the calling terminal can be selected by choosing this argument at runtime. Note, however, that due to fundamental differences in the underlying input routines between Tcl and magic, the terminal-based command entry does not exactly match the original behavior of magic. In particular, the background DRC function does not change the prompt. In addition, "Xterm" consoles must select option "Allow SendEvents" for keystrokes to be echoed from the layout window into the terminal. For security reasons, the application cannot change this option on the terminal.

   (b) `-wrapper` To enforce backward-compatibility, magic appears as it would without the Tcl interpreter (apart from the text entry console) when launched with the same arguments. However, most users will want to make use of the extensions and convenience functions provided by the "wrapper" GUI. These functions are detailed in a separate tutorial (see Tutorial #2).

3. Magic-related programs “ext2spice” and “ext2sim” have been recast as magic commands instead of standalone executables. This makes a good deal of sense, as both programs make heavy use of the magic internal database. Most values required to produce netlists were passed through the intermediary file format `.ext`. However, not all necessary values were included in the `.ext` file format specification, and some of these missing values were hard-coded into the programs where they have since become mismatched to the magic database. This has been corrected so that all netlist output corresponds to the technology file used by a layout.

   As command-line commands, the names have been changed to “exttospice” and “exttosim”, respectively, so that they have correct Tcl syntax. Both commands allow all of the command-line arguments previously accepted by the standalone programs. Some of the more common functions, however, have been duplicated as command options in the usual format of magic commands. For instance, one may use the magic command:

   exttosim help

   to get more information on `exttosim` command-line options.

4. All Tcl interpreter procedures may be freely mixed with magic commands on the command line. For instance, a user can use Tcl to compute arithmetic expressions on the command line:

   copy e [expr{276 * 5 + 4}]

5. A number of magic commands pass values back to the interpreter. For instance, the command

   box values

   returns the lower left- and upper right-hand coordinates of the cursor box in magic internal units. This return value can be incorporated into an expression, such as the one below which moves the box to the left by the value of its own width:

   set bbox [box values]
   set bwidth expr {[lindex $bbox 2] - [lindex $bbox 0]}]
   move e $bwidth

6. Magic prompts for text have been recast as Tk dialog boxes. For instance, the command **writeall** will pop up a dialog box with five buttons, one for each of the available choices (write, flush, skip, abort, and auto).

7. IRSIM is no longer available as a “mode” reached by switching tools by command or the space-bar macro. Instead, IRSIM (version 9.6) can be compiled as a Tcl extension in the same manner as Magic, at compile time. When this is done, IRSIM is invoked simply by typing

irsim

on the Magic command line. IRSIM is loaded as a Tcl package, and IRSIM and Magic commands may be freely mixed on the Magic command line:

assert [getnode]

IRSIM does not require the name of a file to start. Without arguments, it will assume the currently loaded cell is to be simulated, and it will generate the `.sim` file if it does not already exist.

8. In addition to IRSIM, programs **xcircuit** and **netgen** can be compiled as Tcl extensions and provide cabability for schematic capture and layout-vs.-schematic, respectively.
9. Tcl-based Magic makes use of the Tcl variable space. For instance, it keeps track of the installation directory through the variable `CAD_HOME`, which mirrors the value of the shell environment variable of the same name. In addition, it makes use of variables `VDD` and `GND` in the technology file to aid in the extraction of substrate- and well-connected nodes on transistors.
10. Magic input files, such as the `.magic` startup file, are sourced as Tcl scripts, and so may themselves contain a mixture of Tcl and magic commands.

3 Compilation and Installation

Magic is selected for compilation as a Tcl extention when running the

make config

script. The first question asks whether magic should be compiled as a Tcl extension, with the SCHEME interpreter embedded, or as the original version with no interpreter. Subsequent to choosing Tcl as the interpreter and answering the remaining configuration questions, Tcl should be compiled and installed using the commands

make tcl

and

make install-tcl

Note that if magic is to be compiled under different interpreters, it is necessary to perform

make clean

between the compilations, so that all references to the last compiled version are deleted.

4 Dual-Source Input and Backward Compatibility

From its inception, Magic has used an unsual but very effective interface in which commands may be passed to the program from two different sources, the layout window, and the calling terminal. Keystrokes in the layout window are handled by the graphics package. These are interpreted as *macros*. The keystroke values are expanded into magic command-line commands and executed by the command-line command dispatcher routine. Two macros, '`.`' and '`:`' are reserved: The period expands to the last command executed from the command line (*not* from a macro expansion), and the colon initiates a redirection of keystrokes from the layout window into the calling terminal. Magic users quickly get used to the combination of keystrokes, mouse functions, and command-line commands.

In the Tcl-based version of Magic, the command-line dispatching is given over entirely to Tcl, and the dispatching of keystroke events in the layout window is given over entirely to Tk. Unfortunately, in relinquishing these duties, Magic loses some of the effectiveness of its dual-source input model. One aspect of this is that Tcl is a line-based interpreter, and does not recognize anything on the command line until the return key has been pressed and the entire line is passed to the command dispatcher routine. Without any understanding of character-based input, it is difficult to impossible to directly edit the command line from outside the calling terminal, because it is the terminal, and not Tcl, which interprets keystrokes on a character-by-character basis.

The way around this problem is to use a *console*, which is an application that runs in the manner usually expected by Tk, in that it is a GUI-driven application. The interpreter is split into *master* and *slave* interpreters (see the Tcl documentation on the **interp** command), with the master interpreter running the console application and the slave interpreter running the application. The master interpreter then has character-based control over the command line, and the Magic dual-source input model can be implemented exactly as originally designed. The background DRC function can change the command line cursor from '*%*' to '*]*' to indicate that the DRC is in progress, and user input can be redirected from the layout window to the console with the '**:**' keystroke.

In addition to these functions, the console makes use of Tcl's ability to rename commands to recast the basic Tcl output function '**puts**' in such a way that output to `stdout` and `stdin` can be handled differently. In the **TkCon** console, output to `stdout` is printed in blue, while output to `stderr` is printed in red. Magic makes use of Tcl's output procedures so that returned values and information are printed in blue, while warnings and error messages are printed in red. The console also implements command-line history and cut-and-paste methods. The console command-line history replaces the embedded *readline* implementation in magic.

The **TkCon** console is a placeholder for what is intended to be a "project manager" console, with functions more appropriate to the Electronic Design Automation suite of Tcl-based tools. In general, the menu functions which are displayed on the TkCon console are not of particular interest to the Magic user, with the exception of the **History** menu, which can be used to re-enter previously executed commands, and the **Prefs** menu, which includes highlighting options and a very useful calculator mode.

5 Tk Graphics Methods

Because the graphics under Tcl are managed by the Tk package, the only graphics options which can be compiled are the X11 and the OpenGL options. There are numerous differences between these graphics interfaces as they exist under Tcl and under the non-Tcl-based version. The two primary differences are the way windows are generated and the way commands are sent to specific windows. In Tcl-based magic, a layout window does not have to be a top-level application window. by using the command syntax

openwindow *cellname tk_pathname*

An unmapped window can be generated, corresponding to the Tk window hierarchy specified by *tk_pathname* (see the Tk documentation for the specifics of Tk window path name syntax). This window is then mapped and managed by Tk commands. Using this method, a magic window can be embedded inside a "wrapper" application, an example of which has been done with the GUI wrapper invoked with the "`-w`" command-line argument. Extensions of magic window commands have been added so that the wrapper can control the window frame, including the scrollbars and title.

Whenever a window is created, Magic creates a Tcl/Tk command with the same name as the window (this is conventional practice with Tk widgets). Magic and Tcl commands can be passed as arguments to the window command. Such commands are executed relative to the specific window. This applies to all of magic's window-based commands, including instructions such as **move**, **load**, and so forth. Commands which apply to all windows will automatically be sent to all windows. These commands are used primarly by the wrapper GUI, but are also called (transparently to the end user) whenever a command is executed from a layout window via any macro, including the '**:**' command-line entry. In addition, however, they may be called from the command line to perform an action in a specific window. By default (in the absence of a wrapper GUI), magic's windows are named *.magic1*, *.magic2*, and so forth, in order of appearance, and these names are reflected in the title bar of the window. So it is equivalent to do

:move s 10

from layout window '*.magic2*' (where the colon indicates the key macro for command-line entry), and

.magic2 move s 10

typed in the console.

6 Tutorial Examples

Example sessions of running magic in various modes are presented below, along with examples of methods specific to each mode. In the examples below, prompts are shown to indicate the context of each command. The # sign indicates the shell prompt, *(gdb)* indicates the GNU debugger prompt, and *%* indicates the Tcl (i.e., Magic) prompt.

Example 1: Standard Magic Execution

Run Tcl-based magic in its most basic form by doing the following:

```
# magic -noconsole tut2a
```

Magic looks generally like its traditional form, except that the command-line prompt is the Tcl '*%*' prompt. It should be possible to write commands from either the terminal or using the colon keystroke, and it is possible to partially type a command after the colon keystroke and finish the command inside the terminal, but not vice versa. Enabling the colon keystroke may require setting "Allow SendEvents" mode on the calling terminal.

Example 2: Console-based Magic Execution

Run the TkCon console-based magic by doing the following:

```
# magic tut2a
```

The layout window will still look like the usual, basic form. However, the calling terminal will be suspended (unless the application is backgrounded; however, if backgrounding the application, be aware that any output sent to the terminal will hang the application until it is foregrounded) and the Tcl prompt will appear in a new window, which is the **console**. The console may have a slightly different appearance depending on the graphics mode used. For instance, magic has historically had difficulties running in 8-bit (PseudoColor) graphics mode, because it installs its own colormap. Because it does not share the colormap with the calling terminal, the calling terminal gets repainted in random colors from magic's colormap when the cursor is in the layout window. In unlucky setups, text and background may be unreadable.

In the TkCon console setup, the console is mapped prior to determining the graphics mode required, so it also does not share the colormap. However, it is possible to query Magic's colormap to find the location of specific colors, and repaint the text and background of the console accordingly. Thus, the Magic console can be used when the display is in 8-bit PseudoColor mode, without extreme color remappings in the console which make it potentially impossible to read the console when the cursor is in a layout window.

If compiled with both OpenGL and X11 graphics capability, magic will start in X11 mode by default. The OpenGL interface can only be enabled at startup by specifying:

```
# magic -d OGL tut4x
```

The OpenGL interface, in addition to having more solid, vibrant colors, has an additional feature which draws three-dimensional views of a layout. This is discussed in Tutorial #??.

Example 4: The Magic Wrapper GUI

The magic GUI interface is invoked by starting magic with the **-w** option:

```
# magic -w tut2b
```

The immediately noticeable differences are the layer toolbar on the side, and the menu and redesigned title bar on the top. Experimenting with some mouse clicks, the user will note that the magic coordinates of the cursor box are displayed on the right-hand side of the titlebar.

The toolbar contains one example of each layer defined in the magic technology file. Position a box on the screen, then put the cursor over a layer icon and press the middle mouse button. This paints the layer into the box. You will also notice that the name of the layer is printed in the title bar while the cursor is over the layer icon.

The icons have other responses, too. The first and third mouse buttons respectively show and hide the layer in the layout. This works with labels, subcell boundaries, and error paint (the top three layer icons) as well as with regular paintable layers. In addition to mouse button responses, the buttons invoke various commands in reponse to keystrokes. Key ‘p’ paints the layer; key ‘e’ erases it. Key ‘s’ selects the layer in the box while key ‘S’ unselects it.

The menubar has three items, **File**, **Cell**, and **Tech**. Button **File** pops up a menu with options to read and write layout, read and write CIF or GDS output, open a new layout window or close the existing one, or flush the current edit cell. Buttons **Cell** and **Tech** reveal transient windows showing information about the cell hierarchy and the technology file, respectively. The cell hierarchy view is only available if the Tcl/Tk package **BLT** has been compiled and installed on the system. If so, it gives a tree view of the cell hierarchy and allows specific cells to be loaded, edited, and expanded, or viewed to fit the window. This can be expecially useful for querying the cell hierarchy of GDS files, which do not declare top-level cells like CIF files do.

The technology manager window reports the current technology file, its version and description, and the current CIF input and output styles, the current extraction style, and the current value of lambda in microns. The technology file and the CIF input and output styles and extraction style are also buttons which can be used to select a new style or technology from those currently available. Clicking on the current extract style, for instance, gives a list of the styles which have been specified in the current technology file. Clicking on one of the entries in the list makes it the new current extract style.

Cell hierarchy and technology manager windows should be closed by clicking on the “Close” button at the bottom, not by invoking any titlebar functions (which will probably cause the whole application to exit).

Example 5: Batch-mode Magic

Unlike previous versions of magic, it is not necessary to have a layout window present to run magic. Magic may be invoked in “batch mode” by the following command-line:

```
# magic -nowindow tut2b
```

Note however, that most magic commands expect a window to be present to execute the function. The main use for batch mode is for wrapper-type applications to delay opening a layout window until one is requested. For example:

```
# magic -nowindow tut2b
% toplevel .myframe
% openwindow tut2b .myframe.mylayout
% pack .myframe.mylayout
% .myframe.mylayout box 0 0 12 12
% .myframe.mylayout select area poly
% wm withdraw .myframe
% wm deiconify .myframe
% .myframe.mylayout closewindow
```

Note that this is the basic setup of the standard GUI wrapper, albeit with much greater sophistication. The standard GUI wrapper is generated entirely by script, which can be found in the library directory **${CAD_HOME}/lib/magic/tcl/wrapper.tcl**.

Example 6: Tcl-Magic under the Debugger

When running under Tcl, Magic cannot be debugged in the usual manner of executing, for instance, "**gdb magic**", because the main executable is actually the program "**wish**". To run Magic with debugging capability, it is necessary to do the steps below. In the following, it is assumed that Magic has been installed in the default location `CAD_HOME=/usr/local/`, and that the GNU debugger `gdb` is the debugger of choice.

```
# gdb wish
(gdb) run
% source /usr/local/lib/magic/tcl/magic.tcl
     .
     .
     .
% (type Control-C in the terminal window)
(gdb) break TxTclDispatch
(gdb) cont
% paint m1

Breakpoint 1, TxTclDispath (clientData=0x0, argc=2,
     argv=0xbffff400)
     at txCommands.c:1146
1146         DRCBreak();
(gdb)
```

Command-line arguments can be passed to magic through the Tcl variables `argc` and `argv`. The only caveat is that the first character of the first argument is ignored, with the actual parsing starting at the second character of the `argv` list. The integer value `argc` must match the number of entries passed in the `argv` list. For example, do the following:

```
# gdb wish
(gdb) run
% set argc 4
% set argv {–w -d OGL tut1 }
% source /usr/local/lib/magic/tcl/magic.tcl
```

Magic Tcl Tutorial #2: The Wrapper GUI

R. Timothy Edwards

Space Department
Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20723

This tutorial corresponds to Tcl-based Magic version 7.2

1 The Wrapper GUI

Magic Tcl Tutorial #3: Extracting and Netlisting

R. Timothy Edwards

Space Department
Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20723

This tutorial corresponds to Tcl-based Magic version 7.2

1 Tech file extensions for extraction

2 Commands exttospice and exttosim

Magic Tcl Tutorial #4: Simulation with IRSIM

R. Timothy Edwards

Space Department
Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20723

This tutorial corresponds to Tcl-based Magic version 7.2

Tutorials to read first:

Magic Tutorial #1: Getting Started
Magic Tutorial #2: Basic Painting and Selection
Magic Tutorial #4: Cell Hierarchies
Magic Tutorial #8: Circuit Extraction
Magic Tutorial #11: Using IRSIM and RSIM with Magic

Commands introduced in this tutorial:

irsim , getnode, goto
graphnode, watchnode, unwatchnode
movenode, watchtime, unwatchtime, movetime
(plus the standard IRSIM command set)

Macros introduced in this tutorial:

(None)

1 IRSIM Version 9.6

In version 9.6, IRSIM has been redesigned to work under the Tcl interpreter, in the same manner as Magic version 7.2 does. Like Magic version 7.2, section of Tcl as an interpreter is specified at compile-time, along with various use options. The "**make**" method has been rewritten to match the one which Magic uses, so IRSIM can be compiled and installed in a similar manner:

```
make config
make tcl
make install-tcl
```

Tcl-based IRSIM, like its non-interpreter version, can be run as a standalone product, and will simulate a circuit from a `.sim` format file. However, it is specifically designed to be operated in conjunction with magic, with methods for providing feedback directly into the layout from the simulation, and vice versa. There are a number of *cross-application commands*, detailed below, which belong to neither Magic or IRSIM, but are applicable when both are running in the Tcl interpreter at the same time.

The cross-application commands highlight the usefulness of the method of compiling each application as a loadable Tcl object module.

In addition to cross-application commands, Tcl-based IRSIM allows the use of interpreter variables, conditionals, and control structures to set up detailed simulation environments. A random number generator has been added to the Tcl-based version, allowing generation of random bit vectors for statistically-based coverage of input pattern spaces.

2 Invoking IRSIM from Magic

Within the Tcl/Tk environment, IRSIM is easier than ever to invoke. For tutorial purposes, we will use the same cell used for the original Tutorial #11. Unlike the original version, Magic 7.2 requires no preparation for simulation and can operate directly off of the tutorial directory input. Start magic with the command-line

```
# magic -w -d OGL tut11a
```

Note that the OpenGL interface and Wrapper environment specified above are optional, and do not affect the descriptions in this tutorial.

It is not necessary to extract! The scripts which invoke IRSIM are capable of looking for a netlist file to simulate for the currently-loaded cell. Because these exist for the tutorial cells, they will be used. IRSIM is therefore simply invoked by:

```
% irsim
```

You will see a slew of output that looks like the following:

```
Warning:  irsim command ’time’ use fully-qualified name ’::irsim::time
Warning:  irsim command ’start’ use fully-qualified name ’::irsim::sta
Warning:  irsim command ’help’ use fully-qualified name ’::irsim::help
Warning:  irsim command ’path’ use fully-qualified name ’::irsim::path
Warning:  irsim command ’clear’ use fully-qualified name ’::irsim::cle
Warning:  irsim command ’alias’ use fully-qualified name ’::irsim::ali
Warning:  irsim command ’set’ use fully-qualified name ’::irsim::set’
Warning:  irsim command ’exit’ use fully-qualified name ’::irsim::exit
Starting irsim under Tcl interpreter
IRSIM 9.6 compiled on Thu Mar 20 17:19:00 EST 2003
Warning:  Aliasing nodes ’GND’ and ’Gnd’
/usr/local/lib/magic/tutorial/tut11a.sim:  Ignoring lumped-resistance
     (’R’ construct)
```

```
Read /usr/local/lib/magic/tutorial/tut11a.sim lambda:1.00u format:MIT
68 nodes; transistors:  n-channel=56 p-channel=52
parallel txtors:none
%
```

These comments require some explanation. The warning messages all have to do with the fact that certain command names are used both by IRSIM and Magic, or by IRSIM and Tcl or one of its loaded packages (such as Tk). There are several ways to work around the unfortunate consequences of multiply defining command names, but the easiest is to make use of the Tcl concept of *namespaces*. A complete description of Tcl namespaces is beyond the scope of this tutorial; however, a simple description suffices. By prefixing a “scope” to the command, the command can only be executed when the complete name (scope plus the double colon ‘::’ plus the command name) is entered.

In general, the EDA tools make an attempt to allow commands to be entered without the scope prefix at the command line. As long as command names are unique, this is done without comment. However, when commands overlap, the easiest solution is to require the scope prefix. Therefore, the command ‘**set**’ would refer to the Tcl **set** command (i.e., to set a variable), while ‘**irsim::set**’ would refer to the IRSIM command. Some attempt is made to overload commands which conflict but which have unique syntax, so that it is possible to determine which use is intended when the command is dispatched by the interpreter.

In addition to the warnings, there are a few standard warnings about global name aliases and lumped resistance, and some information about the `.sim` file which was read.

3 IRSIM Command Set

In addition to the exceptions noted above for fully-qualified namespace commands, there are several IRSIM commands which are not compatible with Tcl syntax, and these have been renamed. The old and new commands are as follows (see the IRSIM documentation for the full set of commands):

¿	savestate	save network state
¡	restorestate	restore network state
¡¡	restoreall	restore network and input state
?	querysource	get info regarding source/drain connections
!	querygate	get info regarding gate connections
	source *(Tcl command)*	source a command file

Note that the ‘’ command is simply superceded by the Tcl ‘**source**’ command, which is more general in that it allows a mixture of Tcl and IRSIM commands (and commands for any other loaded package, such as Magic) to be combined in the command file.

Once loaded into Tcl alongside Magic via the **irsim** command, the IRSIM commands are typed directly into the Magic command line, and will execute the appropriate IRSIM function. By repeating the contents of Tutorial #11 in the Tcl environment, this method should become clear, as will the benefits of using the interpreter environment for simulation.

To setup the simulation, the equivalent instruction to that of Tutorial #11 is the following:

```
% source ${CAD_HOME}/lib/magic/tutorial/tut11a.cmd
```

Note that because the **source** command is a Tcl command, not a Magic or IRSIM command, it it necessary to specify the complete path to the file, as Tcl does not understand the search path for Magic cells, which includes the tutorial directory.

As most common commands are not among the set that cause conflicts with Magic and Tcl commands, the tutorial command file loads and executes without comment.

Following the example of Tutorial #11, type **c** (IRSIM clock command) on the magic command line to clock the circuit. Values for the watched nodes, which were declared in the tutorial command file, are displayed in the console window. Likewise,

```
h RESET_B hold
```

will set the nodes **`RESET_B`** and **`hold`** to value 1.

4 Feedback to Magic

The cross-application commands reveal the usefulness of having both applications as extensions of the same Tcl interpreter.

While Magic and IRSIM are active and file `tut11a` is loaded, execute the following commands from the command line:

```
stepsize 100
watchnode RESET_B
watchnode hold
```

Note that the nodes and values are immediately printed in the magic window, making use of the magic "**element**" command. These values are persisent in the sense that they will remain through various transformations, openings, and closings of the layout window, but they are temporary in the sense that they will not be saved along with the layout if the file is written (however, this behavior can be modified).

The **watchnode** command requires no special action for placing the label elements in the layout because magic uses the labels or other node information to pinpoint a position in the layout belonging to that node, and places the label element there. It is possible to use **watchnode** with vectors. However, as no location can be pinpointed for a vector, the magic cursor box position will be used to place the label element.

Move the magic cursor box to a empty space in the layout window, and type

```
watchnode bits
```

Now move the cursor box to another empty space and type

```
watchtime
```

Now all of the simulation values of interest are displayed directly on the Magic layout.

The display of any node can be removed with the command **`unwatchnode`**, with the same syntax as **`watchnode`**, and similarly, the display of simulation time can be removed with the command **`unwatchtime`**.

If the position of a label is not in a good position to read, or the relative position of two labels places them on top of one another, making them difficult to read, the labels can be moved using the **movenode** command. For instance, the node `RESET_B` is not exactly on the polysilicon pad. To center it exactly on the pad, select the square pad, so that the box cursor is on it, then do

```
movenode RESET_B
```

The label will be moved so that it is centered on the center of the cursor box. The equivalent method can be applied to the time value using the **`movetime`** command.

It is not necessary to know the name of a node in order to query or display its simulation value. For instance, unexpand the layout of `tut11a.mag`, select an unlabeled node, and use a mixture of IRSIM and magic commands to watch its value:

```
box 93 -104 94 -102
select area
watchnode [getnode]
```

In this example, both the node (`bit_1/tut11d_0/a_39_n23#`) and its value are displayed. Likewise, the **getnode** command can be combined with other IRSIM commands to setup clocks and vectors from unlabeled nodes. This can be particularly useful in situations where it may not be obvious which nodes in a design need to be examined prior to running the simulation.

5 Analyzer Display

Tcl-based IRSIM has a graphical node display which is derived from functions available in the "**BLT**" graphics package. These functions are not particularly well-suited for display of logic values, and so this will probably be replaced in the future with a more appropriate interface. However, it accomplishes most of the functions of the former X11-based analyzer display.

In the Tcl-based IRSIM, no special command is needed to initialize the analyzer display. One command sets up signals to be displayed in the analyzer window. This is:

`graphnode` *name* [*row*] [*offset*]

For display of multiple signals in the window, the optional arguments *row* and *offset* are provided. Each signal which declares a new *row* (default zero) will appear in a separate graph in the display. Signals which appear in the same graph, however, may declare a non-zero *offset* which will set them at a different vertical placement on the graph, for cases in which this provides better viewing than having the signals directly overlapping.

The analyzer display updates at the end of each simulation cycle. Logic values are displayed as 0 or 1, with undefined (value 'X') values displayed as 1/2. Note that the BLT-based interface prohibits the display of multi-bit values, and only nodes, not vectors, can be passed to the **`graphnode`** command.

6 Scripting IRSIM Command Sequences

A consequence of placing IRSIM in an interpreter environment is the ability to use interpreter features such as variables, conditionals, and loops to set up complicated simulation environments.

7 Random Bit Vector Generation

The tutorial examples are small by design, but real systems (such as a microprocessor) are often so complex that generating and simulating an exhaustive set of all possible states of the circuit is impossible, and instead simulations rely on the generation of a set of randomly-generated inputs to test a representative set of states.

Random number generation is not a built-in feature of the Tcl language, but several open-source packages exist, one of which has been incorporated into the IRSIM 9.6 source. The pseudorandom number generator is compiled as a separate Tcl package, but is loaded by the IRSIM startup script. It contains one command, **`random`**, with the following arguments:

`random` *option*

where *option* may be one of:

`-reset` will cause the generator to be reseeded using current pid and current time.

`-seed` *n* will reseed the generator with the integer value *n*.

`-integer` ... will cause the number returned to be rounded down to the largest integer less than or equal to the number which would otherwise be returned.

`-normal` *m s* will cause the number returned to be taken from a gaussian with mean *m* and standard deviation *s*.

`-exponential` *m* will cause the number returned to be taken from an exponential distribution with mean *m*.

`-uniform` *low high* will cause the number returned to be taken from uniform distribution on *[a,b)*.

`-chi2` *n* will cause the number returned to be taken from the chi2 distribution with *n* degrees of freedom.

`-select` *n list* will cause *n* elements to be selected at random from the list *list* with replacement.

`-choose` *n list* will cause *n* elements to be selected at random from the list *list* without replacement.

`-permutation` *n* will return a permutation of $0 \ldots n-1$ if *n* is a number and will return a permutation of its elements if *n* is a list.

The following script clocks a random serial bit vector into a state machine, assuming that **`bit_in`** is the node to set, and that the proper clock vectors have already been set up:

```
for {set i 0} {$i ¡ 100} {incr i} {
        if {[random] ¡ 0.5} {
                        l bit_in
        } else {
```

```
                  h bit_in
         }
         c
     }
```

Magic Tcl Tutorial #5: Writing Tcl Scripts for Magic

R. Timothy Edwards

Space Department
Johns Hopkins University
Applied Physics Laboratory
Laurel, MD 20723

This tutorial corresponds to Tcl-based Magic version 7.2

1 Scripting in Magic

Magic Tutorial #W-1: Design-Rule Extensions

Don Stark

Western Research Laboratory
Digital Equipment Corporation
Palo Alto, CA 94301

This tutorial corresponds to Magic version 7.

Tutorials to read first:

Magic Tutorial #6: Design-Rule Checking
Magic Tutorial #9: Format Conversion for CIF and Calma
Magic Maintainer's Manual #2: The Technology File

Commands introduced in this tutorial:

(None)

Macros introduced in this tutorial:

(None)

1 Introduction

Magic's original design rule checker has proved inadequate to implement all the rules found in advanced technologies. The rules described in this section allow more complicated configurations to be analyzed. Two new rules check a region's area and its maximum width. In addition, width, spacing, area, and maxwidth checks may now be performed on cif layers.

2 Area Rules

The **area** rule is used to check the minimum area of a region. Its syntax is:

area *types minarea minedge why*

Types is a list of types that compose the region, all of which must be on the same plane. *Minarea* is the minimum area that a region must have, while *minedge* is the minimum length of an edge for the region. This second dimension is basically an optimization to make the design rule checker run faster; without it, the checker has to assume that a region 1 lambda wide and *minarea* long is legal, and it must examine a much larger area when checking the interaction between cells. Specifying *minedge* reduces this interaction distance. An example rule is:

area (emitter,em1c)/npoly 6 2 "emitter must be at least 2x3"

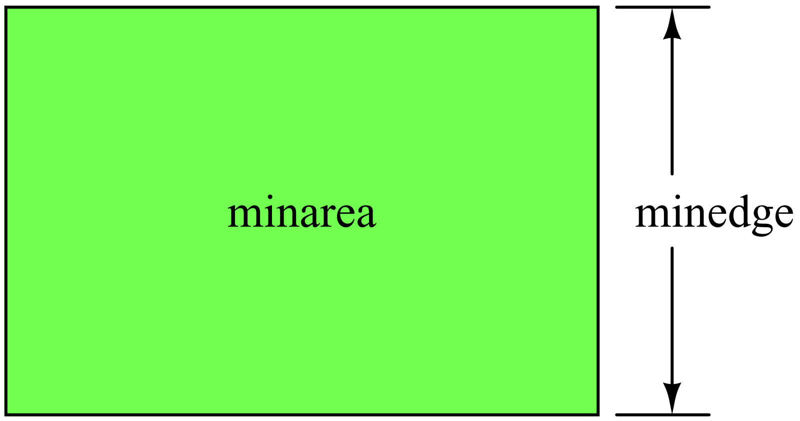


Figure 1: Example of the area rule.

3 Maxwidth Rules

Sometimes a technology requires that a region not be wider than a certain value. The **maxwidth** rule can be used to check this.

maxwidth *layers mwidth bends why*

Layers, the types that compose the region, must all be in the same plane. The region must be less than *mwidth* wide in either the horizontal or vertical dimension. *Bends* takes one of two values, **bend_illegal** and **bend_ok**. For **bend_illegal** rules, the checker forms a bounding box around all contiguous tiles of the correct type, then checks this box's width. For example:

**maxwidth (emitter,em1c)/npoly 2 bend_illegal \
"emitter width cannot be over 2"**

bend_ok rules are used to check structures where the region must be locally less than maxwidth, but may contain bends, T's, and X's.

maxwidth trench 2 bend_ok "trench must be exactly 2 wide"

Warning: the bend_ok rule is basically a kludge, and may fail for regions composed of more than one type, or for intersections more complicated than T's or X's. Figure 3 shows some examples of both types of rules.

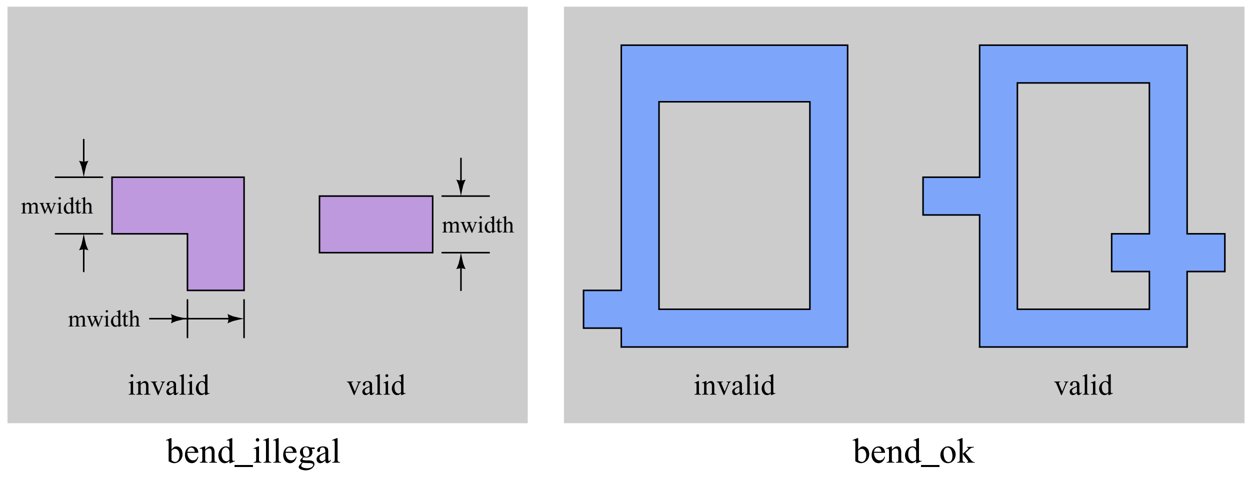


Figure 2: Examples of the maxwidth rule. The dogleg at the left would be ok in a **bend_ok** rule, but fails in a **bend_illegal** one, where the region's bounding box is checked. For **bend_ok** rules, each tile in the region is checked. The left shape fails in two places: the top horizontal part is too thick and the stub at the bottom intersects the region in a shape other than a T or X.

4 Rules on CIF layers

For technologies with complicated generated layers, it is often difficult to check design rules on the abstract types that are drawn in Magic. To ameliorate this problem, the extended checker allows simple checks to be performed on cif layers. The rules that can be checked are width, spacing, area, and maxarea. Since checking rules on the cif layers requires that these layers be generated, these checks are considerably slower than the normal ones, and should only be used when absolutely necessary.

4.1 Setting the CIF style

The **cifstyle** rule is used to select which **cifoutput** style is used.

cifstyle *cif_style*

Cif_style must be one of the cif styles included in the cifoutput section. In the current implementation, the cif checker generates all the layers in the style regardless of whether they are actually used in design-rule checks; for speed, defining a separate cif style for design rule checking it may be worthwhile when only a few layers are checked. Any layer in the cif style, defined by either a *layer* or a *templayer* rule, may be checked.

4.2 Width Checks

The syntax for **cifwidth** is analogous to that of the regular width rule:

cifwidth *layer width why*

Layer is a single cif layer. (To do width checks with more than one cif layer, **or** all the layers into a new *templayer*). *Width* is the minimum width of the region in centimicrons.

4.3 Spacing Checks

The **cifspacing** rule is also very similar to the regular rule:

cifspacing *layer1 layer2 separation adjacency why*

Layer1 and *layer2* are both cif layers. If *adjacency* is **touching_ok**, then layer1 must equal layer2. For **touching_illegal** rules, *layer1* and *layer2* may be any two cif layers. *Separation* is given in centimicrons.

4.4 Area Checks

The area rule is:

cifarea *layer minarea minedge why*

Layer is again a single cif layer. *minedge* is expressed in centimicrons, and *minarea* is given in square centimicrons.

4.5 Maxwidth Checks

The maxwidth rule is:

cifmaxwidth *layer mwidth bends why*

Again, *layer* is a single cif layer, and *mwidth* is given in centimicrons.

Magic Technology Manual #2: Scalable CMOS

Shih-Lien Lu

Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90291

John Ousterhout

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

1. Introduction

NOTE: This manual is no longer maintained by the Magic team, as MOSIS has taken over responsibility for it. For the latest copy, send an electronic mail message to "mosis@mosis.edu" with the following lines:

REQUEST: INFORMATION
TOPIC: SCMOS_MANUAL.INF
NET-ADDRESS: <*put your e-mail address here*>
REQUEST: END

The Net-Address line is optional -- leave it out if you aren't sure of your e-mail address. In almost all cases MOSIS can figure out your return address.

The latest technology file is also available. Send a message similar to the above message, but request topic SCMOS.TECH instead of SCMOS_MANUAL.

Magic Technology Manual #1: NMOS

John Ousterhout

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

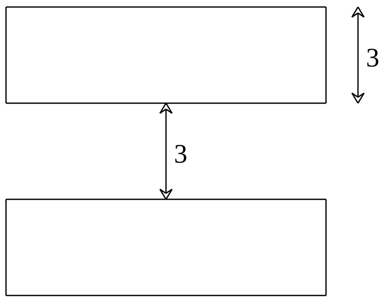
(Warning: Process details often change. Contact MOSIS or your fab line to verify information in this document.)

1. Introduction

This document describes Magic's NMOS technology. It includes information about the layers, design rules, routing, CIF generation, and extraction. This technology is available by the name **nmos** (run Magic with the shell command **magic -T nmos**). The design rules described here are for the standard Mead and Conway NMOS process with butting contacts omitted and buried contacts added. There is a single layer each of metal and polysilicon. If you've been reading the Mead and Conway text, or if you've already done circuit layout with a different editing system, don't forget that these are not the layers that actually end up on masks. Contacts and transistors are drawn in a stylized form that omits implants, vias, and buried windows.

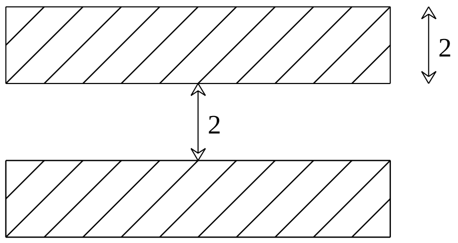
2. Layers and Design Rules

2.1. Metal



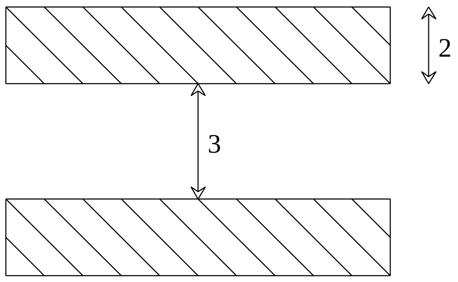
There is only one layer of metal, and it is drawn in blue. Magic accepts the names **metal** or **blue** for this layer. Metal must always be at least 3 units wide and must be separated from other metal by at least 3 units.

2.2. Polysilicon



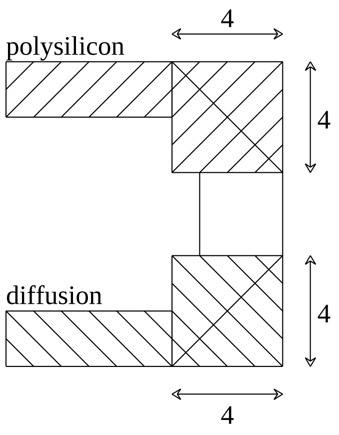
Polysilicon is drawn in red, and can be referred to in Magic as either **polysilicon** or **red**. It has a minimum width of 2 units and a minimum spacing of 2 units.

2.3. Diffusion



Diffusion is drawn in green, and can be referred to in Magic as either **diffusion** or **green**. It has a minimum width of 2 units and a minimum spacing of 3 units.

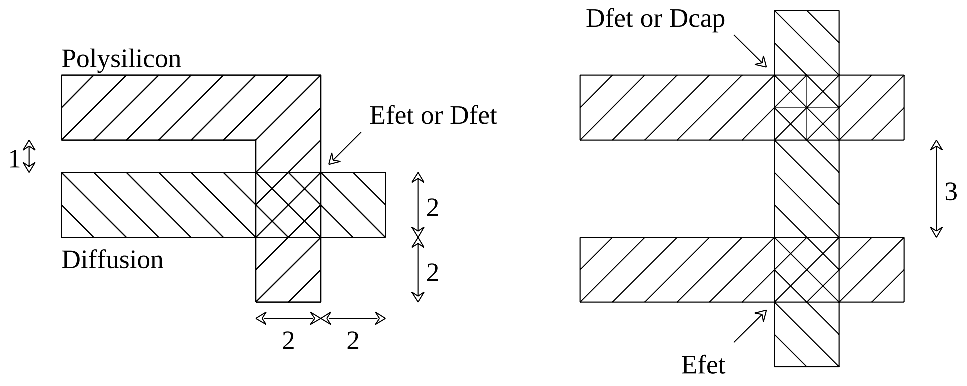
2.4. Contacts to Metal



Contacts between metal and polysilicon, and between metal and diffusion, have similar forms. Poly-metal contacts can be referred to as **pmc** or **poly_metal_contact**; they are drawn to look like metal running on top of poly, with an ‘‘X’’ over the area of the contact. Diffusion-metal contacts are similar, except that they look like metal running on top of diffusion, and have names **dmc** and **diff_metal_contact**. Contacts are drawn differently in Magic than they will appear in the CIF: you do *not* draw the via hole. Instead, you draw the outer area of the metal pad around the contact, which must be at least 4 units on each side. Magic will fill in the appropriate via when CIF is generated. If you draw contacts larger than 4 units on a side, Magic will fill in as many 2-by-2 CIF via holes (with 2-unit spacings) as it can. Contacts areas must be rectangular in shape: contacts of the same type may not abut.

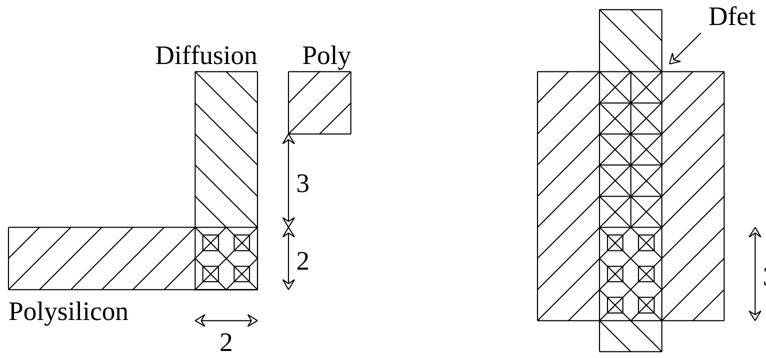
An additional kind of contact, called **glass_contact**, is used to generate holes in the overglass layer for use in bonding to pads. This layer is drawn as gray stripes over blue, and includes both metal and the overglass hole.

2.5. Transistors



There are three transistor structures in the NMOS technology. Enhancement transistors are known by the names **efet** and **enhancement_fet**, and are drawn to look like red over green, with green stripes. You get efet automatically when you paint poly over diffusion or vice versa. Depletion transistors are known by the names **dfet** and **depletion_fet**, and are drawn the same way, except with yellow stripes. A third type of material is called **depletion_capacitor** or **dcap**. It is displayed with yellow crosses over the transistor area, and is identical to dfet except that there are no overhang design rules for it since it is assumed to be used only as a capacitor. You do not drawn any implants in Magic, but just use a different material for the transistor. Magic will generate the implants automatically. All transistors must be at least 2 units on each side, and there must be a poly or diffusion overhang for 2 units on each side of efet or dfet (this is not required for dcap). Poly must be separated from diffusion by at least one unit except where it is forming a transistor. Dfet and dcap must be at least 3 units from efet in order to keep the implant from contaminating the enhancement transistor.

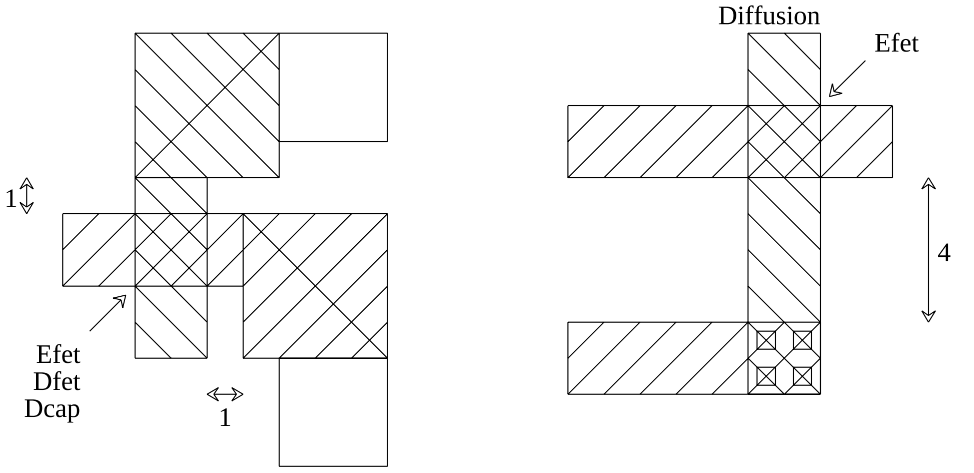
2.6. Buried Contacts



Buried contacts go by the names **bc** and **buried_contact**. They are drawn in a brownish color (the same as transistors), except with solid black squares over their area. As with other contacts, you draw just the area where the two connecting materials (poly and diffusion) overlap; Magic will generate the CIF buried window, which is actually larger than the overlap area. Buried contacts come in two forms. The normal form is 2 units on a side, and no poly or diffusion overhang is required. The second form is used only next to depletion transistors, and is a 3-by-2 structure abutting the depletion transistor. This form is a little controversial, since it results in larger-than-normal variations in the size of the depletion transistor. As a consequence, Magic reports design-rule violations wherever buried contacts abut depletion transistors less than 4 units long. In butting bc-dfet structure, you should measure the transistor length from the bc-dfet boundary.

WARNING: there is one additional rule for buried contacts that is NOT enforced by Magic. Where diffusion enters a buried contact, there must be no unrelated polysilicon for 3 units on that side of the buried contact. This rule is necessary because the buried window extends outward from the buried contact by one unit on the diffusion side, and polysilicon must be far enough away to avoid shorting to the diffusion through the buried window. Unfortunately, there is no way to check this rule in Magic without being extremely conservative (the rule would have to require no poly whatsoever on the diffusion side, even if the poly was connected to the buried contact). So, for now, this rule is not checked. Be careful!

2.7. Transistor Spacings



Transistors must be spaced at least 1 unit from any contact to metal, in order to keep the contact from shorting the transistor. In addition, buried contacts must be at least 4 units from enhancement transistors in the diffusion direction. This rule applies only to the side of buried contact where diffusion leaves the contact.

2.8. Hierarchical Constraints

The design-rule checker enforces several constraints on how subcells may overlap. The general rule is that overlaps may be used to connect portions of cells, but the overlaps must not change the structure of the circuit. Thus, for example, it is acceptable for poly in one cell to overlap poly-metal contact in another cell, but it is not acceptable for poly in one cell to overlap diffusion in another (thereby forming a transistor).

For contacts, there are additional restrictions. A contact in one cell may not overlap a contact in any other cell unless the two contacts have same type and they occupy exactly the same area. Partial overlaps are not permitted, nor are abutting contacts of the same type (contacts of different types may abut, as long as the abutment doesn't violate any other design rules). The contact restrictions are necessary to guarantee that CIF can be generated correctly in a hierarchical fashion.

3. Routing

If you use Magic's automatic routing tools on an NMOS design, the routing will be run in metal and polysilicon, with metal as the primary layer. The routing will be placed on a 7-unit grid.

4. Reading and Writing CIF

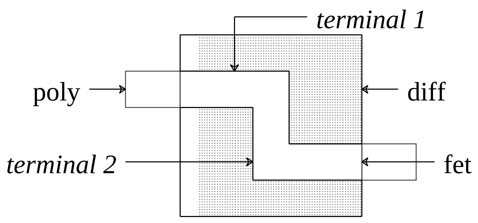
There is only one CIF output style available in the NMOS technology: **lambda=2**. The CIF layers in this style, and their meanings, are:

Name	Meaning
NP	polysilicon
ND	diffusion
NM	metal
NI	depletion implant: generated around depletion transistors and depletion contacts
NC	contact via: generated as small squares inside poly-metal contacts and diffusion-metal contacts
NB	buried window: generated around buried contacts
NG	overglass via: generated for overglass contacts

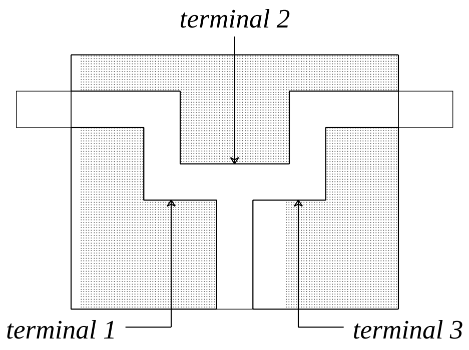
To see exactly where each CIF layer is generated for a particular design, use the **:cif see** command. There is also just one CIF input style. It is called **lambda=2** and can be used to read files written by Magic in the **lambda=2** style, or files written by Caesar using the standard NMOS technology with a scale factor of 200.

5. Extraction

Transistors of type **efet** or **dfet** in the NMOS technology must have at least two diffusion terminals. A diffusion terminal is a contiguous region along the perimeter of the transistor channel that connects to diffusion, as shown below:



2 diff terminals



3 diff terminals

A transistor may have more than two diffusion terminals, in which case it is modeled as a collection of two-terminal transistors. If only one diffusion terminal is present, the the extractor flags this as an error and outputs a transistor with the source and drain shorted together.

Transistors of the special type **dcap** may have as few as one diffusion terminal. Although their normal use is as capacitors, the extractor will output them as though they were a **dfet**. It is up to simulation programs to compute the capacitance of a **dcap** from the area and perimeter of its channel.

The NMOS technology file currently contains little information on parasitic coupling capacitances. As a result, overlap capacitance, and sidewall overlap capacitance will always be zero.