September 1990

WRL Research Report 90/7

1990 DECWRL/ Livermore Magic Release

Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

1990 DECWRL/Livermore Magic Release

**Robert N. Mayo, Michael H. Arnold, Walter S. Scott,
Don Stark, Gordon T. Hamachi**

September, 1990

Prepared with the assistance of:

Digital Equipment Corporation
Western Research Laboratory
Palo Alto, California

Lawrence Livermore National Labs
"O" Division
Livermore, California

Stanford University
Center for Integrated Systems
Palo Alto, California

University of California
Department of EECS
Berkeley, California

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

Table of Contents

Overview of the DECWRL/Livermore Magic Release

This document corresponds to Magic version 6.

1. Introduction

This version of Magic, version 6, gathers together work done by numerous people at several institutions since Magic version 4 was released from Berkeley on the 1986 VLSI tools tape. This is a release of Magic and IRSIM only. You'll probably want to obtain other tools by ordering the 1986 VLSI Tools Tape from Berkeley.

This release has been prepared with the assistance of several groups. Much of the new software came from Walter Scott's group at the Lawrence Livermore National Labs (LLNL). LLNL also provided partial funding to help prepare the release. Digital Equipment Corporation's Western Research Lab (DECWRL) helped out by providing computer equipment, a place to work, and the services of one of us (Robert Mayo). Don Stark, Michael Arnold, and Gordon Hamachi also worked on the release at DECWRL. Stanford donated significant pieces of new code, including a simulation system called IRSIM. Other individuals and institutions have also contributed code and assistance in ways too numerous to detail here.

New features in Magic Version 6 include:

New and Improved Routing - *Michael Arnold and Walter Scott of LLNL*

> Three major routing improvements have been made in this version of Magic. There is a new, improved, global router courtesy of Walter Scott (of LLNL). Walter Scott has also added a gate array router. See the "garoute" command in the manual page for details. Michael Arnold (of LLNL) has written an interactive maze router that allows the user to specify hints to control the routing. See the documentation for the "iroute" command.

Extractor Enhancements - *Don Stark of Stanford and Walter Scott of LLNL*

> The new "extresis" command, developed by Don Stark, provides substantially better resistance extraction. Magic's normal extraction ("extract") lumps resistances on a node into a single value. In branching networks, this approximation is often not acceptable. Resis was written to solve this problem. Walter Scott added accurate path length extraction, an important feature when dealing with high speed circuits, such as ECL.

New contact structure - *Walter Scott and Michael Arnold of LLNL and Don Stark of Stanford*

Multilayer contacts are handled better. In the previous version of Magic, there needed to be a separate contact type for each possible combination of contact layers over a given point. This caused a combinatorial explosion of tile types for multi-layer technologies with stacked contacts. Under the new scheme, there are only a couple of tile types for each layer: one that connects up, one that connects down, and one that connects in both directions.

Simulator Interface to IRSIM - *Stanford*

A simulator interface is provided courtesy of Stanford. See the commands "startrsim", "simcmd", and "rsim". The irsim simulator, Stanford's much improved rewrite of esim, is included in this distribution. Credit goes to Mike Chow, Arturo Salz, and Mark Horowitz.

New device/machine Support - *Various*

X11 is fully supported in this release, and is the preferred interface. Older drivers for graphics terminals and X10 are also included, but X11 is the preferred interface (meaning it is better supported and you'll have lots of company). Magic's X11 driver has a long history, starting with an X10 driver by Doug Pan at Stanford. Brown University, the University of Southern California, the University of Washington, and Lawrence Livermore National Labs all prepared improved versions, some of them for X11. Don Stark of Stanford took on the task of pulling these together and producing the X11 driver in this release.

Magic runs on a number of workstations, such as the DECstation 3100 and Sun's SPARC processors. Partial Unix System V support is provided, via the compilation flags mentioned below. The system also runs on the MacII. Don Stark gets credit for the System V mods and support for HP machines, while Mike Chow helped get it running on the MacII.

To assist people with small machines (such as the Mac II), Magic can now be compiled without some of its fancy features. Compilation flags are provided, as indicated below, to eliminate things like routing, plotting, or calma output. This is courtesy of Don Stark.

Reorganization of Magic Source Directory

Magic, as previously distributed, was set up with the assumption that lots of people would be changing the code at the same time. As a result, the makefiles did all sorts of paranoid things like making extra copies of the source code whenever a module was re-installed.

Since Magic is more stable now, this copying is no longer needed. Instead, each makefile invokes the script **../:instclean** after installing a module. This script, by default, doesn't copy the source code but does leave the .o files around. This cuts down on the disk space needed by a factor of two. You can change the script if you want the copying, or if you want to delete unused .o files to save even more disk space.

Lots of bug fixes - *Various*

Lots of bugs have been fixed in this release. We'd like to thank everybody that has reported bugs in the past. If you find a new bug, please report it

as mentioned below.

2. Distribution Information

This version of Magic is available via FTP. Contact "magic@decwrl.dec.com" for information.

For a handling fee, this version of Magic may be obtained on magnetic tape from:

EECS/ERL Industrial Liaison Program
479 Cory Hall
University of California at Berkeley
Berkeley, CA 94720

3. Bug Reports

Maintenance of Magic is a volunteer effort. Please send descriptions of bugs via InterNet e-mail to "magic@decwrl.dec.com" or via Uucp e-mail to "decwrl!magic". If you develop a fix for the problem, please send that too!

4. Changes for Magic maintainers

Previous releases of Magic expected to find their system files in the home directory of the user **cad**. The default behavior of version 6 is no different, but it is possible to put the files in another directory by setting the **CAD_HOME** shell environment variable. If this variable is set, magic will use that location instead of the ~cad it finds in the password file.

4.1. INSTALLING MAGIC

The distribution tape contains a version of Magic ready to run on Digital's line of Ultrix RISC workstations, such as the DECstation 3100. For other machines, read ahead. In any event, all users should set their shell environment variable CAD_HOME to point to the place where the tape is loaded, unless that place is ~cad, in which case things will default correctly.

Before installing Magic, you should set your shell environment variable CAD_HOME to point to the place where you loaded the tape. If you "cd" to the magic source directory (${CAD_HOME}/src/magic) you will find a makefile. A "**make config**" will run a configuration script that asks questions about your configuration and sets up magic to be compiled for your local environment.

After running a "make config", you can run a "**make force**" to force a complete recompilation of magic. A "**make install**" will then copy the binaries to the ${CAD_HOME}/bin area, as well as install things in ${CAD_HOME}/lib and ${CAD_HOME}/man.

Included in this documentation is a set of Magic maintainer's manuals. These should be read by anybody interested in modifying Magic or by anybody that is having difficulty installing it on their system.

4.2. Technology file changes

Users of Magic 4 should have little trouble switching to Magic 6.

A new section, the **mzrouter** section needs to be added to your technology files. See the mzrouter section of the tutorial *Magic Maintainer's Manual #2: The Technology File* for details.

Display styles must be defined in the *.tech* file for the mzrouter hint layers magnet, fence and rotate. We suggest copying this information from the styles section of the scmos technology file on the distribution tape. You'll also need to include these display styles in your *.dstyle* file.

5. Beta-test Sites

We'd like to thank the beta-test sites that tried out this version of Magic, reported bugs and fixes in a timely manner, and ported the code to new machines:

Mike Chow, Apple Computer
Arun Rao, Arizona State University
Richard Hughey, Brown University
Rick Carley, Carnegie-Mellon University
Hank Walker, Carnegie-Mellon University
Christos Zoulas, Cornell University
Andreas Andreou, John Hopkins University
George Entenman, The Microelectronics Center of North Carolina
Shih-Lien Lu, The MOSIS Service
Jen-I Pi, The MOSIS Service
Guntram Wolski, Silicon Engineering, Inc.
Don Stark, Stanford University
Gregory Frazier, University of California at Los Angeles
Yuval Tamir, University of California at Los Angeles
Steven Parkes, University of Illinois
Larry McMurchie, University of Washington
Tim Heldt, Washington State University
David Lee, Xerox Palo Alto Research Center

Martin Harriman of Silicon Engineering wrote a "select less" command for Magic during the beta-test phase. "Select less" has been a much-requested feature.

In addition to the persons named above, there were many other beta-test users of Magic at these and other sites -- too many to list here. We appreciate their help. We also acknowledge the help of the pre-release sites, who tested a version that included most of the fixes from the beta-test phase.

NAME

ext2dlys – create a SCALD wire-delays file from a tree of .ext files

SYNOPSIS

ext2dlys [**–d** *psPerPf*] [**–l** *psPerCentimicron*] [**–m** *minmult maxmult*] [**–o** *outfile*] [**–t** *capscale*] [**–D** *drivefile*] [**–I** *iload*] [**–L** *netfile*] [**–M** *scaldmapfile*] [**–O** *oload*] [*extcheck-options*] *file*

DESCRIPTION

Ext2dlys is used to produce a SCALD wire-delays file (in *dlys* (5) format) on standard output, to be used in simulation and timing verification. It computes the wire delay information from capacitance in the circuit extracted from a layout by *magic* (1).

The filename *file* given to *ext2dlys* is the name of the root **.ext** file of the extracted circuit, and also of the **.net** file. The **.ext** files, in *ext* (5) format, contain the capacitance to substrate for each electrical node, specify the connectivity of the circuit, and also give distance information. The **.net** file, in *net* (5) format, lists the nets and terminals in the circuit that will be present in the **.dlys** file. All terminals in the **.net** file are by default considered to be inputs (receivers) unless explicitly identified as drivers in the *drivefile* given with the **–D** option; see the description below. In addition to identifying the terminals of interest, the **.net** file gives the signal name associated with each net as a comment line immediately prior to the list of terminals in the net.

The remaining arguments to *ext2dlys* tell how this capacitance and distance information is to be converted into delay, as well as specifying the use of alternate files:

–d *psPerPf*

Used to turn capacitance into delay; one picofarad is equal to *psPerPf* picoseconds of delay. The default value is **100.0**, or roughly what one would expect if using 100 ohm drivers. The value of *psPerPf* is used only for drivers whose effective on resistance hasn't been given explicitly in the *drivefile* specified with the **–D** flag (see below).

–l *psPerCentimicron*

Used to turn distance into delay; one centimicron of distance is equal to *psPerCentimicron* picoseconds of delay.

–m *minmult maxmult*

Multipliers to convert estimated delays into best-case (*minmult*) and worst-case (*maxmult*). Both are **1.0** by default.

–o *outfile*

Write the output to *outfile* (note that no suffix is implied) instead of to the standard output.

–t *capscale*

Gives a scale factor by which units of capacitance in the **.sim** file will be multiplied in order to give femtofarads. *Capscale* may be a real number; its default value is **1.0**.

–D *drivefile*

Also used to turn capacitance into delay, but on a per-net basis. Each line of the file *drivefile* consists of a hierarchical pin name (of an output driver) and its associated ''drive factor'' (equivalent to *delay* in the **–d** flag above), namely the number of picoseconds per picofarad for the net driven by that output pin. Nets driven by a pin listed not in this file use the default delay specified by **–d** above. If this file isn't given, we don't know for certain which pins are the drivers in each net, so we arbitrarily pick one pin per net and assume it is the driver.

–I *iload* In addition to the capacitance reported in the **.sim** file for each net, add an additional *iload* attofarads of capacitance for each input on a given net to the total capacitance for that net. (Inputs are counted only if they appear in the **.net** file.) The default value of *iload* is **0.0**, since it varies so much from one technology to the next. This option is provided to account for extra transistor capacitance not computed by the extractor, such as when the technology of the circuit being extracted is non-MOS (e.g, bipolar).

−L *netfile*

Instead of using *file***.net** as the netlist file, use *netfile***.net** instead.

−M *scaldmap*

If specified, then *scaldmap* is read (note no suffix implied) to obtain a translation between Magic terminal names and SCALD pin names. Each line in *scaldmap* contains a Magic name followed by a SCALD name. The Magic name is terminated by the first blank; the SCALD name continues from the next non-blank character to the end of the line, possibly including embedded blanks. When writing the output file, the corresponding SCALD name is used instead of the Magic name for each pin in a net. See *dlys* (5) for more details of the output file format.

−O *oload*

In addition to the capacitance reported in the **.sim** file for each net, add an additional *oload* attofarads of capacitance for each output on a given net to the total capacitance for that net. The default value of *oload* is **0.0**. If only **−I** and not **−O** is specified, *ext2dlys* treats this as though both **−I** and **−O** had been specified with the same values; inputs and outputs are not distinguished.

In addition, all of the options of *extcheck* (1) are accepted.

SEE ALSO

extcheck (1), ext2sim (1), ext2spice (1), magic (1), dlys (5), ext (5)

AUTHOR

Walter Scott

NAME

ext2sim – convert hierarchical *ext* (5) extracted-circuit files to flat *sim* (5) files

SYNOPSIS

ext2sim [**–a** *aliasfile*] [**–l** *labelsfile*] [**–o** *simfile*] [**–A**] [**–B**] [**–F**] [**–L**] [**–t**] [*extcheck-options*] [*-y num*] [*-f mit|lbl|su*] [*-J hier|flat*] [*-j device:sdRclass[/subRclass]/defaultSubstrate*] *root*

DESCRIPTION

Ext2sim will convert an extracted circuit from the hierarchical *ext* (5) representation produced by Magic to the flat *sim* (5) representation required by many simulation tools. The root of the tree to be extracted is the file *root***.ext**; it and all the files it references are recursively flattened. The result is a single, flat representation of the circuit that is written to the file *root***.sim**, a list of node aliases written to the file *root***.al**, and a list of the locations of all nodenames in CIF format, suitable for plotting, to the file *root***.nodes**. The file *root***.sim** is suitable for use with programs such as *crystal* (1), *esim* (1), or *sim2spice* (1).

The following options are recognized:

–a *aliasfile* Instead of leaving node aliases in the file *root***.al**, leave it in *aliasfile*.

–l *labelfile* Instead of leaving a CIF file with the locations of all node names in the file *root***.nodes**, leave it in *labelfile*.

–o *outfile* Instead of leaving output in the file *root***.sim**, leave it in *outfile*.

–A Don't produce the aliases file.

–B Don't output transistor or node attributes in the **.sim** file. This option will also disable the output of information such as the area and perimeter of source and drain diffusion and the fet substrate. For compatibitlity reasons the latest version of ext2sim outputs this information as node attibutes. This option is necessary when preparing input for programs that don't know about attributes, such as *sim2spice* (1) (which is actually made obsolete by *ext2spice* (1), anyway), or *rsim* (1).

–F Don't output nodes that aren't connected to fets (floating nodes).

–L Don't produce the label file.

–t*char* Trim characters from node names when writing the output file. *Char* should be either "#" or "!". The option may be used twice if both characters are desired.

–f *MIT|LBL|SU* Select the output format. MIT is the traditional *sim*(5) format. LBL is a variant of it understood by *gemini*(1) which includes the substrate connection as a fourth terminal before length and width. SU is the internal Stanford format which is described also in *sim*(5) and includes areas and perimeters of fet sources, drains and substrates.

–y *num* Select the precision for outputing capacitors. The default is 1 which means that the capacitors will be printed to a precision of .1 fF.

–J *hier|flat* Select the source/drain area and perimeter extraction algorithm. If *hier* is selected then the areas and perimeters are extracted *only within each subcell*. For each fet in a subcell the area and perimeter of its source and drain within this subcell are output. If two or more fets share a source/drain node then the total area and perimeter will be output in only one of them and the other will have 0. If *flat* is selected the same rules apply only that the scope of search for area and perimeter is the whole netlist. In general *flat* (which is the default) will give accurate results (it will take into account shared sources/drains) but hier is provided for backwards compatibility with version 6.4.5. On top of this selection you can individually control how a terminal of a specific fet will be extracted if you put a source/drain attribute. *ext:aph* makes the extraction for that specific terminal hierarchical and *ext:apf* makes the extraction flat (see the magic tutorial about attaching attribute labels). Additionaly to ease extraction of bipolar transistors the gate attribute

ext:aps forces the output of the substrate area and perimeter for a specific fet (in flat mode only).

−j *device:sdRclass[/subRclass]/defaultSubstrate*
Gives ext2sim information about the source/drain resistance class of the fet type *device*. Makes *device* to have *sdRclass* source drain resistance class, *subRclass* substrate (well) resistance class and the node named *defaultSubstrate* as its default substrate. The defaults are nfet:0/Gnd and pfet:1/6/Vdd which correspond to the MOSIS technology file but things might vary in your site. Ask your local cad administrator.

The way the extraction of node area and perimeter works in magic the total area and perimeter of the source/drain junction is summed up on a single node. That is why all the junction areas and perimeters are summed up on a single node (this should not affect simulation results however).

Special care must be taken when the substrate of a fet is tied to a node other than the default substrate (eg in a bootstraping charge pump). To get the correct substrate info in these cases the fet(s) with separate wells should be in their own separate subcell with ext:aph attributes attached to their sensitive terminals (also all the transistors which share sensistive terminals with these should be in another subcell with the same attributes).

In addition, all of the options of *extcheck* (1) are accepted.

SCALING AND UNITS

If all of the **.ext** files in the tree read by *ext2sim* have the same geometrical scale (specified in the **scale** line in each **.ext** file), this scale is reflected through to the output, resulting in substantially smaller **.sim** files. Otherwise, the geometrical unit in the output **.sim** file is a centimicron.

Resistance and capacitance are always output in ohms and femptofarads, respectively.

SEE ALSO

extcheck (1), ext2dlys (1), ext2spice (1), magic (1), rsim (1), ext (5), sim (5)

AUTHOR

Walter Scott additions/fixes by Stefanos Sidiropoulos.

BUGS

Transistor gate capacitance is typically not included in node capacitances, as most analysis tools compute the gate capacitance directly from the gate area. The **-c** flag therefore provides a limit only on non-gate capacitance. The areas and perimeters of fet sources and drains work only with the simple extraction algorith and not with the extresis flow. So you have to model them as linear capacitors (create a special extraction style) if you want to extract parasitic resistances with extresis.

NAME

ext2spice − convert hierarchical *ext* (5) extracted-circuit files to flat *spice* files

SYNOPSIS

ext2spice [**−B**] [*extcheck-options*] [*-M|m*] [*-y num*] [*-f hspice|spice3|spice2*] [*-J hier|flat*] [*-j device:sdRclass[/subRclass]/defaultSubstrate*] *root*

DESCRIPTION

Ext2spice will convert an extracted circuit from the hierarchical *ext* (5) representation produced by Magic to a flat spice file which can be accepted by spice2, spice3, hspice and other simulation tools. The root of the tree to be extracted is the file *root***.ext**; it and all the files it references are recursively flattened. The result is a single, flat representation of the circuit that is written to the file *root***.spice** .

The following options are recognized:

−o *outfile* Instead of leaving output in the file *root***.spice**, leave it in *outfile*.

−B Don't output transistor or node attributes in the spice file. Usually the attributes of a node or a device are output as special comments **fetattr and **nodeatrr which can be processed further to create things such a initial conditions etc.

−F Don't output nodes that aren't connected to fets (floating nodes). Normally capacitance from these nodes is output with the comment **FLOATING attached on the same line.

−t*char* Trim characters from node names when writing the output file. *Char* should be either "#" or "!". The option may be used twice if both characters are desired. Trimming "#" and "!" is enabled by default when the format is hspice.

-*M|m* Merge parallel fets. *-m* means conservative merging of fets that have equal widths only (usefull with hspice format multiplier if delta W effects need to be taken care of). -M means aggresive merging: the fets are merged if they have the same terminals and the same length.

−y *num* Select the precision for outputing capacitors. The default is 1 which means that the capacitors will be printed to a precision of .1 fF.

−f *hspice|spice2|spice3*
Select the output format. Spice3 is the the format understood by the latest version of berkeley spice. Node names have the same names as they would in a *sim*(5) file and no special constructs are used. Spice2 is the format understood by the older version of spice (which usually has better convergence). Node names are numbers and a dictionary of number and corresponding node is output in the end. HSPICE is a format understood by meta-software's hspice and other commercial tools. In this format node names cannot be longer than 15 characters long (blame the fortran code): so if a hierarchical node name is longer it is truncated to something like x1234/name where x1234 is an alias of the normal node hierarchical prefix and name its hierarchical postfix (a dictionary mapping prefixes to real hierarchical paths is output at the end of the spice file). If the node name is still longer than 15 characters long (again blame the fortran code) it is translated to something like z@1234 and the equivalent name is output as a comment. In addition since hspice supports scaling and multipliers so the output dimensions are in lambdas and if parallel fets are merged the hspice construct *M* is used.

−J *hier|flat* Select the source/drain area and perimeter extraction algorithm. If *hier* is selected then the areas and perimeters are extracted *only within each subcell*. For each fet in a subcell the area and perimeter of its source and drain within this subcell are output. If two or more fets share a source/drain node then the total area and perimeter will be output in only one of them and the other will have 0. If *flat* is selected the same rules apply only that the scope of search for area and perimeter is the whole netlist. In general *flat* (which

is the default) will give accurate results (it will take into account shared sources/drains) but hier is provided for backwards compatibility with version 6.4.5. On top of this selection you can individually control how a terminal of a specific fet will be extracted if you put a source/drain attribute. *ext:aph* makes the extraction for that specific terminal hierarchical and *ext:apf* makes the extraction flat (see the magic tutorial about attaching attribute labels). Additionaly to ease extraction of bipolar transistors the gate attribute *ext:aps* forces the output of the substrate area and perimeter for a specific fet (in flat mode only).

–j *device:sdRclass[/subRclass]/defaultSubstrate*

Gives ext2sim information about the source/drain resistance class of the fet type *device*. Makes *device* to have *sdRclass* source drain resistance class, *subRclass* substrate (well) resistance class and the node named *defaultSubstrate* as its default substrate. The defaults are nfet:0/Gnd and pfet:1/6/Vdd which correspond to the MOSIS technology file but things might vary in your site. Ask your local cad administrator.

The way the extraction of node area and perimeter works in magic the total area and perimeter of the source/drain junction is summed up on a single node. That is why all the junction areas and perimeters are summed up on a single node (this should not affect simulation results however).

Special care must be taken when the substrate of a fet is tied to a node other than the default substrate (eg in a bootstraping charge pump). To get the correct substrate info in these cases the fet(s) with separate wells should be in their own separate subcell with ext:aph attributes attached to their sensitive terminals (also all the transistors which share sensistive terminals with these should be in another subcell with the same attributes).

In addition, all of the options of *extcheck* (1) are accepted.

The awk filter spice2sim is provided with the current distribution for debugging purposes.

SEE ALSO

extcheck (1), ext2spice (1), magic (1), rsim (1), ext (5), sim (5)

AUTHOR

Stefanos Sidiropoulos.

BUGS

The areas and perimeters of fet sources and drains work only with the simple extraction algorith and not with the extresis flow. So you have to model them as linear capacitors (create a special extraction style) if you want to extract parasitic resistances with extresis.

NAME

extcheck − check hierarchical *ext* (5) files for global node connectivity and summarize number of fets, nodes, etc.

SYNOPSIS

extcheck [**−c** *cthresh*] [**−p** *path*] [**−r** *rthresh*] [**−s** *sym=value*] [**−C**] [**−R**] [**−S** *symfile*] [**−T** *tech*] *root*

DESCRIPTION

Extcheck will read an extracted circuit in the hierarchical *ext* (5) representation produced by Magic, check to ensure that all global nodes (those to which a label ending in an exclamantion point is attached) are fully connected in the layout, and then print a count of the number of various items (nodes, fets, etc) encountered while flattening the circuit. The root of the tree to be processed is the file *root***.ext**; it and all the files it references are recursively flattened.

The following options are recognized:

−c *cthresh*
Set the capacitance threshold to *cthresh* femtofarads. *Extcheck* will count the number of explicit internodal capacitors greater than *cthresh*, the number of nodes whose capacitance is greater than *cthresh*, as well as the total number of nodes. (Other programs such as *ext2sim* (1) use this option as a threshold value below which a capacitor will not be output). The default value for *cthresh* is 10 femtofarads.

−p *path* Normally, the path to search for **.ext** files is determined by looking for **path** commands in first ~cad/lib/magic/sys/.magic, then ~/.magic, then .magic in the current directory. If **−p** is specified, the colon-separated list of directories specified by *path* is used instead. Each of these directories is searched in turn for the **.ext** files in a design.

−r *rthresh*
Set the resistance threshold to *rthresh* ohms. Similar in function to **−c**, but for resistances. The default value for *rthresh* is 10 ohms.

−s *sym=value*
It's possible to use special attributes attached to transistor gates to control the length and width of transistors explicitly, rather than allowing them to be determined by the extractor. These attributes are of the form **ext:w=***width*^ or **ext:l=***length*^, where *width* or *length* can either be numeric, or textual. (The trailing ''^'' indicates that these are transistor gate attributes). If textual, they are treated as symbols which can be assigned a numeric value at the time *ext2sim* is run. The **−s** flag is used to assign numeric values to symbols. If a textual symbol appears in one of the above attributes, but isn't given a numeric value via **−s** (or **−S** below), then it is ignored; otherwise, the transistor's length or width is set to the numeric value defined for that symbol. *(This option is not currently used by extcheck, but it is common to ext2sim (1) and other tools that are written using the extflat (3) library)*

−C Set the capacitance threshold to infinity. Because this avoids any internodal capacitance processing, all tools will run faster when this flag is given.

−R Set the resistance threshold to infinity.

−S *symfile*
Each line in the file *symfile* is of the form *sym=value*, just like the argument to the **−s** flag above; the lines are interpreted in the same fashion. *(This option is not currently used by extcheck, but it is common to ext2sim et. al.)*

−T *tech* Set the technology in the output **.sim** file to *tech*. This overrides any technology specified in the root **.ext** file.

SEE ALSO

ext2dlys (1), ext2sim (1), ext2spice (1), magic (1), rsim (1), sim2spice (1), ext (5), sim (5)

AUTHOR

Walter Scott

BUGS

The **–s** mechanism is incomplete; it should allow quantities other than transistor lengths and widths to be specified.

NAME

fsleeper – run sleeper remotely

SYNOPSIS

fsleeper [**–t** *tty*] [**–l** *user*] [*remotemachine*]

DESCRIPTION

Fsleeper is used if you wish to run a program such as *magic* (1) on a different machine (*remotemachine*) than the one to which a graphics terminal is attached, and the local graphics terminal has no login process.

Normally, *fsleeper* will start a remote sleeper on the companion graphics terminal for your terminal. This graphics terminal is found by looking in the file ~cad/lib/displays, as described in *displays* (5). If a different graphics terminal is desired, it may be specified by the **-t** flag. Note that this is the terminal on the local machine, not the remote machine. (The remote terminal will be printed by *sleeper* (1) when it starts up on the remote machine).

Also, normally *fsleeper* will attempt to log in as the user **sleeper** on the remote machine. If a different user name is desired, it may be specified with the **-l** flag. This user name must exist on *remotemachine*.

FILES

~cad/lib/displays

SEE ALSO

magic(1), rsleeper(1), sleeper(1), displays(5)

AUTHOR

Walter Scott

BUGS

If no *remotemachine* is specified, it defaults to **ucbkim**. This is fine for Berkeley, but useless elsewhere.

NAME

grSunProg – internal process for Magic's Sun 120 display driver

SYNOPSIS

grSunProg *colorWindowName textWindowName notifyPID requestFD pointFD buttonFD*

DESCRIPTION

GrSunProg is an internal program used by Magic when using the Sun 120 workstation's display. This manual page is intended only for Magic maintainers.

GrSunProg collects button pushes from the color window and sends them over a pipe to Magic. The program also responds to requests from Magic for the mouse position. In addition, this program tells Suntools to forward characters typed in the color window directly to Magic's text window.

ARGUMENTS

All six arguments are required:

colorWindowName

This is the name of the color window that magic is running under (such as **/dev/win3**). Magic normally opens up the color monitor with a single, large, window on it.

textWindowName

This is the name of the text window that contains Magic's command log. Keyboard events are forwarded to this window.

notifyPID

If this processID is not 0, then SIGIO signals are sent to this process when there is data for it.

requestFD pointFD buttonFD

These are the file descriptors that grSunProg should use in its interface (see below). They are small integers printed as strings.

INTERFACE

Button pushes are sent out over file descriptor *buttonFD*. A button push is encoded as two characters followed by two integers giving the location of the button push. The first character is either 'L', 'M', or 'R' depending on the button pushed: Left, Middle, or Right. The next character is either 'D' or 'U' depending on the action: Up or Down. The two numbers are the X and Y coordinates of the button push. This string is followed by a newline. Example: **LD 123 342** means that the left button was pushed down at location (123, 342).

GrSunProg sometimes receives a character from Magic over file descriptor *requestFD*. If this character is an EOF, then the program terminates. If this character is an 'A', then grSunProg responds with a 'P' and the current mouse coordinates over file descriptor *pointFD*. This string is followed by a newline. Example: **P 101 23** means that the mouse is currently at location (101, 23).

SEE ALSO

magic(1) grsunprog2(1)

AUTHOR

Robert N. Mayo

NAME

magic – VLSI layout editor

SYNOPSIS

magic [**–T** *technology*] [**–d** *device_type*] [**–g** *graphics_port*] [**–m** *monitor_type*] [**–i** *tablet_port*] [**–D**] [**–F** *object_file save_file*] [*file*]

DESCRIPTION

Magic is an interactive editor for VLSI layouts that runs under 4.3 BSD Unix, as well as derivatives such as Digital's Ultrix and Sun's SunOS. This man page is a reference manual; if you are a first-time user, you should use the Magic tutorials in ''The 1989 Livermore/DEC-WRL Magic Release'' to get acquainted with the system.

Magic runs in two different configurations. For workstations with an integrated color display, one window of the screen is used to display text (commands and Magic's responses) and other windows are used for displaying layouts in color. In older systems using serial-line terminals, Magic uses two terminals: one for text and a separate color display for displaying layouts. In these systems you should run Magic from the text display.

Normally, Magic is run under a window system such as X11 (preferred) or Sun Tools. The command line switch "-d" can be used to tell Magic which kind of window system you are running, although Magic is pretty good at guessing. When using serial-line terminals, the "-d", "-g", and "-i" switches can be used, and the file ~cad/lib/displays should be created by the system administrator (see DISPLAYS(5) manual page).

Here are the options accepted by Magic:

–T The next argument is the name of a technology. The tile types, display information, and design rules for this technology are read by Magic from a technology file when it starts up. The technology defaults to ''scmos''.

–d The next argument is the type of workstation or graphics display being used. Magic supports these types:

NULL A null device for running Magic without using a graphics display.

X11 X-windows, version 11 release 3. The is the preferred interface. Magic acts as a client to the X window server and interfaces to all graphics terminals supported by the X server. The window manager must be configured to pass mouse buttons, without interpretation, to clients. It is recommended that the meta key be used with mouse buttons to communicate requests to the window manager. Standard window manager commands manipulate Magic windows.

Addition information on Magic's X11 driver, including options for .Xdefaults files, may be found in ''Magic Maintainer's Manual #4: Using Magic Under X Windows''.

X10 An X driver for X version 10. Currently not used much, and being phased out.

XWIND

Simply another name for either the X11 driver or the X10 driver. This is normally set to refer to whichever driver is more popular at a given site. This is here mostly for backward compatibility reasons.

The following drivers are used on Suns when X is not available.

SUN60 A Sun Microsystems workstation model Sun3/60C. (May work for Sun4/60C, also.) The Sun60 display is the same as a Sun110 display, as far as Magic is concerned.

SUN110

A Sun Microsystems workstation, model Sun3/110C (color display). You must be running Suntools. This is virtually identical to the SUN160 display type below. May also work for Sun4/110C.

SUN160

A Sun 160 workstation with a single screen that has 8 bit-planes of color. You must be running Suntools in order for Magic to run with this option. Also, you can not resize or redisplay Magic windows while Magic is collecting a command (since Magic is only one process). Because Sun's window package doesn't do interrupt processing, you can't interrupt Magic unless you point to its text window.

SUNBW

A black & white Sun workstation. Because this system only has one bit-plane, Magic does extra redisplay whenever it erases the box or highlight areas. Also, it is hard to see all the layers since they are drawn as stippled areas instead of colored areas. You must be running Suntools in order for Magic to run with this option, and the caveats for the SUN160 version also apply to this version.

The following drivers are available, but are seldom used and thus may be in disrepair.

UCB512

An old AED512 with the Berkeley microcode ROMs and an attached bitpad (SummaGraphics Bitpad One). The ROMs are available from AED.

UCB512N

A new "Colorware" AED512 with the Berkeley microcode ROMs and an attached bitpad (SummaGraphics Bitpad One). The ROMs are available from AED. We do not recommend the GTCO bitpad, since we have heard that their Summagraphics emulation mode can't handle up/down button encoding nor double button pushes.

AED767

An AED767 with a SummaGraphics Mouse. Because of missing features in this device, programmable cursors and Bit-Blt do not work. Many thanks to Norm Jouppi and DECWRL for porting Magic to this device.

UCB1024

An AED1024 with a SummaGraphics Mouse and AED's Magic microcode ROMs that implement the same operations as the UCB512 ROMs. Thanks to LSI Logic for this port.

AED1024

An AED1024 with a SummaGraphics Mouse and rev. D roms. Because of missing features in this device, programmable cursors do not work. Many thanks to Peng Ang and LSI Logic Corp. for porting Magic to this device.

other AEDs

Other AEDs can be handled by modifying Magic's file grAed1.c. There are just too many combinations of options for AEDs for us to be able to supply drivers for all of them.

SUN120

A Sun Microsystems workstation, model Sun2/120 with the SunColor option (/dev/cgone0) and the Sun optical mouse. Also works on some old Sun1s with the 'Sun2 brain transplant'. You must be running Suntools on the black and white display.

If no device is specified, Magic tries to guess which driver is appropriate (by checking the environment variables and by poking around in /dev). Types listed in ~cad/lib/displays override the default type.

–g The next argument is the name of the device to use for communication with the graphics display. This is usually of the form **/dev/tty***xx* (for displays connected by RS232 lines, such as the AED family) or for some workstations, the name of the frame buffer device.

–m The next argument is used to select the right color map for the monitor's phosphors. ''Std'' works well for most monitors. This option is seldom used anymore, now that monitors are manufactured

with more consistent phosphors.

–i The next argument is the name of the port to use for input from the tablet. This defaults to whatever port is being used for the graphics output, and thus only needs to be specified under unusual circumstances. Only used for serial-line graphics terminals, not workstations.

–D (System maintainers only). Run Magic in ''debug'' mode. This is intended for use with debuggers such as *dbx* (1) which would otherwise catch the **SIGIO** signal that is sent to Magic on each keystroke. When running Magic in debug mode, keystrokes and mouse clicks won't interrupt the background design-rule checker, so it's generally best to run with design-rule checking disabled (**:drc off**). When **-D** is set, crashes do not generate mail to the system maintainer and coredumps are not created.

–F (This switch only works on VAXes, and hasn't been tested recently.) The next two arguments are filenames. The first, *object_file*, is the name of the file that was executed to run this version of Magic. The second, *save_file*, is the name of a new file. After performing all initializations (reading in the technology file, loading the style information and colormap, etc), an executable image of Magic is stored in *save_file*. This executable image may then be run as a normal Magic, except that it starts up much more quickly. The symbol table from *object_file* is copied to *save_file* so the new version can be debugged.

When Magic starts up it looks for a command file in ~cad/lib/magic/sys/.magic and processes it if it exists. Then Magic looks for a file with the name ''.magic'' in the home directory and processes it if it exists. Finally, Magic looks for a .magic file in the current directory and reads it as a command file if it exists. The .magic file format is described under the **source** command.

COMMANDS -- GENERAL INFORMATION

Magics uses three sorts of commands. Pressing a mouse button is one sort of command. You can also enter commands by typing a **:** or **;** character followed by the text of the command. Multiple commands may be specified on one line by separating them with semicolons. The third command form consists of single-letter abbreviations called ''macros''; macros are invoked by pressing single keys without typing a **:** first. Certain macros are predefined in the systemwide ~cad/lib/magic/sys/.magic file, but you can override them and add your own macros using the **macro** command (described below under COMMANDS FOR ALL WINDOWS).

Most commands deal with the window underneath the cursor, so if a command doesn't do what you expect make sure that you are pointing to the correct place on the screen. There are several different kinds of windows in Magic (layout, color, and netlist); each window has a different command set, described in a separate section below.

MOUSE BUTTONS FOR LAYOUT WINDOWS

Magic uses a three button mouse. The buttons are interpreted in a way that depends on the current tool, as indicated by the shape of the cursor (see the **tool** command). The various tools are described below. The initial tool is **box**. These interpretations apply only when mouse buttons are pressed in the interior of a layout window.

Box Tool

This is the default tool, and is indicated by a crosshair cursor. It is used to position the box and to paint and erase:

left This button is used to move the box by one of its corners. Normally, this button picks up the box by its lower-left corner. To pick the box up by a different corner, click the right button while the left button is down. This switches the pick-up point to the corner nearest the cursor. When the button is released, the box is moved to position the corner at the cursor location. If the box has been set to snap to the window's grid (see the **:snap** command), then the box corner is left aligned with the grid that the user has chosen for

the window with the **:grid** command, even if that grid is not visible.

right Change the size of the box by moving one corner. Normally this button moves the upper-right corner of the box. To move a different corner, click the left button while the right button is down. This switches the corner to the one nearest the cursor. When you release the button, three corners of the box move in order to place the selected corner at the cursor location (the corner opposite the one you picked up remains fixed). Snapping to the window's grid is handled as for the left button.

middle (bottom)
Used to paint or erase. If the crosshair is over paint, then the area of the box is painted with the layer(s) underneath the crosshair. If the crosshair is over white space, then the area of the box is erased.

Wiring Tool
The wiring tool, indicated by an arrow cursor, is used to provide an efficient interface to the wiring commands:

left Same as the long command **wire type**.

right Same as the long command **wire leg**.

middle (bottom)
Same as the long command **wire switch**.

Netlist Tool
This tool is used to edit netlists interactively. It is indicated by a thick box cursor.

left Select the net associated with the terminal nearest the cursor.

right Find the terminal nearest the cursor, and toggle it into the current net (if it wasn't already in the current net) or out of the current net (if it was previously in the net).

middle (bottom)
Find the terminal nearest the cursor, and join its net with the current net.

Rsim Tool
Used when running the IRSIM simulator under Magic. A pointing hand is used as the cursor.

left Moves the box just like the box tool.

right Moves the box just like the box tool.

middle (bottom)
Displays the Rsim node values of the selected paint.

LONG COMMANDS FOR LAYOUT WINDOWS

These commands work if you are pointing to the interior of a layout window. Commands are invoked by typing a colon (''':'') or semi-colon (''';''), followed by a line containing a command name and zero or more parameters. In addition, macros may be used to invoke commands with single keystrokes. Useful default macros are set in the global **.magic** file (in **~cad/lib/magic/sys** unless the **CAD_HOME** environment variable is set). You can list all current macros with the **macro** command, described under ''LONG COMMANDS FOR ALL WINDOWS''. Unique abbreviations are acceptable for all keywords in commands. The commands are:

addpath *searchpath*
Add more directories to the end of Magic's cell search path. See the documentation for the **path** command for an explanation of the search path.

array *xsize ysize*
Make many copies of the selection. There will be *xsize* instances in the x-direction and *ysize* instances in the y-direction. Paint and labels are arrayed by copying them. Subcells are not copied, but instead each instance is turned into an array instance with elements numbered from 0

to *xsize*-1 in the x-direction, and from 0 to *ysize*-1 in the y-direction. The spacing between elements of the array is determined by the box x- and y-dimensions.

array *xlo ylo xhi yhi*
Identical to the form of **array** above, except that the elements of arrayed cells are numbered left-to-right from *xlo* to *xhi* and bottom-to-top from *ylo* to *yhi*. It is legal for *xlo* to be greater than *xhi*, and also for *ylo* to be greater than *yhi*.

box [*args*]
Used to change the size of the box or to find out its size. There are several sorts of arguments that may be given to this command:

(No arguments.)
Show the box size and its location in the edit cell, or root cell of its window if the edit cell isn't in that window.

direction [*distance*]
Move the box *distance* units in *direction*, which may be one of **left**, **right**, **up**, or **down**. *Distance* defaults to the width of the box if *direction* is **right** or **left**, and to the height of the box if *direction* is **up** or **down**.

width [*size*]

height [*size*]
Set the box to the width or height indicated. If *size* is not specified the width or height is reported.

x1 y1 x2 y2
Move the box to the coordinates specified (these are in edit cell coordinates if the edit cell is in the window under the cursor; otherwise these are in the root coordinates of the window). *x1* and *y1* are the coordinates of the lower left corner of the box, while *x2* and *y2* are the upper right corner. The coordinates must all be integers.

calma [*option*] [*args*]
This command is used to read and write files in Calma GDS II Stream format (version 3.0, corresponding to GDS II Release 5.1). This format is like CIF, in that it describes physical mask layers instead of Magic layers. In fact, the technology file specifies a correspondence between CIF and Calma layers. The current CIF output style (see **cif ostyle**) controls how Calma stream layers are generated from Magic layers. If no arguments are given, the **calma** command generates a Calma stream file for the layout in the window beneath the cursor in *file***.strm**, where *file* is the name of the root cell. This stream file describes the entire cell hierarchy in the window. The name of the library is the same as the name of the root cell. *Option* and *args* may be used to invoke one of several additional operations:

calma flatten
Ordinarily, Magic arrays are output using the Calma ARRAY construct. After a **calma flatten** command, though, arrays will be output instead as a collection of individual cell uses, as occurs when writing CIF.

calma help
Print a short synopsis of all of the **calma** command options.

calma labels
Output labels whenever writing a Calma output file.

calma lower
Allow both upper and lower case to be output for label text. This is the default behavior; **calma nolower** causes lower case to be converted to upper case on output.

calma noflatten

Undoes the effect of a prior **:calma flatten** command, re-enabling the output of Magic arrays using the Calma ARRAY construct.

calma nolabels
Don't output labels when writing a Calma output file.

calma nolower
Convert lower to upper case when outputting labels.

calma read *file*
The file *file*.**strm** is read in Calma format and converted to a collection of Magic cells. The current CIF input style determines how the Calma layers are converted to Magic layers. The new cells are marked for design-rule checking. Calma format doesn't identify the root of the collection of cells read in, so none of the cells read will appear on the display; use **load** to make them visible. If the Calma file had been produced by Magic, then the name of the root cell is the same as the library name printed by the **:calma read** command.

calma write *fileName*
Writes a stream file just as if no arguments had been entered, except that the output is written into *fileName*.**strm** instead of using the root cell name for the file name.

channels
This command will run just the channel decomposition part of the Magic router, deriving channels for the area under the box. The channels will be displayed as outlined feedback areas over the edit cell.

cif [*option*] [*args*]
Read or write files in Caltech Intermediate Form (CIF). If no arguments are given, this command generates a CIF file for the window beneath the cursor in *file*.**cif**, where *file* is the name of the root cell. The CIF file describes the entire cell hierarchy in the window. *Option* and *args* may be used to invoke one of several additional CIF operations:

cif arealabels [**yes | no]**
Enables/disables the cif area-label extension. If enabled, area labels are written via the **95** cif extension. If disabled, labels are collapsed to points when writing cif and the **94** cif construct is used. Area-labels are disabled by default (many programs don't understand cif area-labels).

cif help Print a short synopsis of all of the cif command options.

cif istyle [*style*]
Select the style to be used for CIF input. If no *style* argument is provided, then Magic prints the names of all CIF input styles defined in the technology file and identifies the current style. If *style* is provided, it is made the current style.

cif ostyle [*style*]
Select the style to be used for CIF output. If no *style* argument is provided, then Magic prints the names of all CIF output styles defined in the technology file and identifies the current style. If *style* is provided, it is made the current style.

cif read *file*
The file *file*.**cif** is read in CIF format and converted to a collection of Magic cells. The current input style determines how the CIF layers are converted to Magic layers. The new cells are marked for design-rule checking. Any information in the top-level CIF cell is copied into the edit cell. Note: this command is not undo-able (it would waste too much space and time to save information for undoing).

cif see *layer*
In this command *layer* must be the CIF name for a layer in the current output style.

Magic will display on the screen all the CIF for that layer that falls under the box, using stippled feedback areas. It's a bad idea to look at CIF over a large area, since this command requires the area under the box to be flattened and therefore is slow.

cif statistics
Prints out statistics gathered by the CIF generator as it operates. This is probably not useful to anyone except system maintainers.

cif write *fileName*
Writes out CIF just as if no arguments had been entered, except that the CIF is written into *fileName***.cif** instead of using the root cell name for the file name. The current output style controls how CIF layers are generated from Magic layers.

cif flat *fileName*
Writes out CIF as in the **cif write** command, but flattens the design first (e.g. creates an internal version with the cell hierarchy removed). This is useful if one wishes to use the **and-not** feature of the CIF output styles, but is having problems with interactions of overlapping cells.

clockwise [*degrees*]
Rotate the selection by **90**, **180** or **270** degrees. After the rotation, the lower-left corner of the selection's bounding box will be in the same place as the lower-left corner of the bounding box before the rotation. *Degrees* defaults to **90**. If the box is in the same window as the selection, it is rotated too. Only material in the edit cell is affected.

copy [*direction* [*amount*]]

copy to *x y*
If no arguments are given, a copy of the selection is picking up at the point lying underneath the box lower-left corner, and placed so that this point lies at the cursor position. If *direction* is given, it must be a Manhattan direction (e.g. **north**, see the ''DIRECTIONS'' section below). The copy of the selection is moved in that direction by *amount*. If the box is in the same window as the selection, it is moved too. *Amount* defaults to **1**. The second form of the command behaves as though the cursor were pointing to (*x*, *y*) in the edit cell; a copy of the selection is picked up by the point beneath the lower-left corner of the box and placed so that this point lies at (*x*, *y*).

corner *direction1 direction2* [layers]
This command is similar to **fill**, except that it generates L-shaped wires that travel across the box first in *direction1* and then in *direction2*. For example, **corner north east** finds all paint under the bottom edge of the box and extends it up to the top of the box and then across to the right side of the box, generating neat corners at the top of the box. The box should be at least as tall as it is wide for this command to work correctly. *Direction1* and *direction2* must be Manhattan directions (see the section DIRECTIONS below) and must be orthogonal to each other. If *layers* is specified then only those layers are used; otherwise all layers are considered.

delete Delete all the information in the current selection that is in the edit cell. When cells are deleted, only the selected use(s) of the cell is (are) deleted: other uses of the cell remain intact, as does the disk file containing the cell. Selected material outside the edit cell is not deleted.

drc option [*args*]
This command is used to interact with the design rule checker. *Option* and *args* (if needed) are used to invoke a **drc** command in one of the following ways:

drc catchup
Let the checker process all the areas that need rechecking. This command will not return until design-rule checking is complete or an interrupt is typed. The checker will run even if the background checker has been disabled with **drc off**.

drc check

Mark the area under the box for rechecking in all cells that intersect the box. The recheck will occur in background after the command completes. This command is not normally necessary, since Magic automatically remembers which areas need to be rechecked. It should only be needed if the design rules are changed.

drc count
Print the number of errors in each cell under the box. Cells with no errors are skipped.

drc find [*nth*]
Place the box over the *nth* error area in the selected cell or edit cell, and print out information about the error just as if **drc why** had been typed. If *nth* isn't given (or is less than 1), the command moves to the next error area. Successive invocations of **drc find** cycle through all the error tiles in the cell. If multiple cells are selected, this command uses the upper-leftmost one. If no cells are selected, this command uses the edit cell.

drc help
Print a short synopsis of all the drc command options.

drc off Turn off the background checker. From now on, Magic will not recheck design rules immediately after each command (but it will record the areas that need to be rechecked; the command **drc on** can be used to restart the checker).

drc on Turn on the background checker. The checker will check whatever modifications have not already been checked. From now on, the checker will reverify modified areas as they result from commands. The checker is run in the background, not synchronously with commands, so it may get temporarily behind if massive changes are made.

drc printrules [*file*]
Print out the compiled rule set in *file*, or on the text terminal if *file* isn't given. For system maintenance only.

drc rulestats
Print out summary statistics about the compiled rule set. This is primarily for use in writing technology files.

drc statistics
Print out statistics kept by the design-rule checker. For each statistic, two values are printed: the count since the last time **drc statistics** was invoked, and the total count in this editing session. This command is intended primarily for system maintenance purposes.

drc why
Recheck the area underneath the box and print out the reason for each violation found. Since this command causes a recheck, the box should normally be placed around a small area (such as an error area).

dump *cellName* [**child** *refPointC*] [**parent** *refPointP*]
Copy the contents of cell *cellName* into the edit cell so that *refPointC* in the child is positioned at point *refPointP* in the edit cell. The reference points can either be the name of a label, in which case the lower-left corner of the label's box is used as the reference point, or as a pair of numbers giving the (*x*, *y*) coordinates of a point explicitly. If *refPointC* is not specified, the lower-left corner of *cellName* cell is used. If *refPointP* is not specified, the lower-left corner of the box tool is used (the box must be in a window on the edit cell). After this command completes, the new information is selected.

edit Make the selected cell the edit cell, and edit it in context. The edit cell is normally displayed in brighter colors than other cells (see the **see** command to change this). If more than one cell is selected, or if the selected cell is an array, the cursor position is used to select one of those cells as the new edit cell. Generally, Magic commands modify only the current edit cell.

erase [*layers*]

For the area enclosed by the box, erase all paint in *layers*. (See the ''LAYERS'' section for the syntax of layer lists). If *layers* is omitted it defaults to ***,labels**. See your technology manual, or use the **layers** command, to find out about the available layer names.

expand [**toggle**]

If the keyword **toggle** is supplied, all of the selected cells that are unexpanded are expanded, and all of the selected cells that are expanded are unexpanded. If **toggle** isn't specified, then all of the cells underneath the box are expanded recursively until there is nothing but paint under the box.

extract *option* [*args*]

Extract a layout, producing one or more hierarchical **.ext** files that describe the electrical circuit implemented by the layout. The current extraction style (see **extract style** below) determines the parameters for parasitic resistance, capacitance, etc. that will be used. The **extract** command with no parameters checks timestamps and re-extracts as needed to bring all **.ext** files up-to-date for the cell in the window beneath the crosshair, and all cells beneath it. Magic displays any errors encountered during circuit extraction using stippled feedback areas over the area of the error, along with a message describing the type of error. Option and *args* are used in the following ways:

extract all

All cells in the window beneath the cursor are re-extracted regardless of whether they have changed since last being extracted.

extract cell *name*

Extract only the currently selected cell, placing the output in the file *name*. If more than one cell is selected, this command uses the upper-leftmost one.

extract do [*option*]

extract no *option*

Enable or disable various options governing how the extractor will work. Use **:extract do** with no arguments to print a list of available options and their current settings. When the **adjust** option is enabled, the extractor will compute compensating capacitance and resistance whenever cells overlap or abut; if disabled, the extractor will not compute these adjustments but will run faster. If **capacitance** is enabled, node capacitances to substrate (perimeter and area) are computed; otherwise, all node capacitances are set to zero. Similarly, **resistance** governs whether or not node resistances are computed. The **coupling** option controls whether coupling capacitances are computed or not; if disabled, flat extraction is significantly faster than if coupling capacitance computation is enabled. Finally, the **length** option determines whether or not pathlengths in the root cell are computed (see **extract length** below).

extract help

Prints a short synopsis of all the **extract** command options.

extract length [*option args*]

Provides several options for controlling which point-to-point path lengths are extracted explicitly. The extractor maintains two internal tables, one of *drivers*, or places where a signal is generated, and one of *receivers*, or places where a signal is sent. The components of each table are hierarchical label names, defined by means of the two commands **extract length driver** *name1* [*name2* ...] and **extract length receiver** *name1* [*name2* ...]. If extraction of pathlengths is enabled (''**:extract do length**''), then when the root cell in an extract command is being extracted, the extractor will compute the shortest and longest path between each driver and each receiver on the same electrical net, and output it to the **.ext** file for the root cell. Normally, one should create a file of these Magic commands for the circuit drivers and receivers of interest, and use **source** to

read it in prior to circuit extraction. **Extract length clear** removes all the entries from both the driver and receiver tables.

extract parents
Extract the currently selected cell and all of its parents. All of its parents must be loaded in order for this to work correctly. If more than one cell is selected, this command uses the upper-leftmost one.

extract showparents
Like **extract parents**, but only print the cells that would be extracted; don't actually extract them.

extract style [*style*]
Select the style to be used for extraction parameters. If no *style* argument is provided, then Magic prints the names of all extraction parameter styles defined in the technology file and identifies the current style. If *style* is provided, it is made the current style.

extract unique [**#**]
For each cell in the window beneath the cursor, check to insure that no label is attached to more than one node. If the **#** keyword was not specified, whenever a label is attached to more than one node, the labels in all but one of the nodes are changed by appending a numeric suffix to make them unique. If the **#** keyword is specified, only names that end in a ''**#**'' are made unique; any other duplicate nodenames that don't end in a ''**!**'' are reported by leaving a warning feedback area. This command is provided for converting old designs that were intended for extraction with Mextra, which would automatically append unique suffixes to node names when they appeared more than once.

extract warn [[**no**] *option* | [**no**] **all**]
The extractor always reports fatal errors. This command controls the types of warnings that are reported. *Option* may be one of the following: **dup**, to warn about two or more unconnected nodes in the same cell that have the same name, **fets**, to warn about transistors with fewer than the minimum number of terminals, and **labels**, to warn when nodes are not labeled in the area of cell overlap. In addition, **all** may be used to refer to all warnings. If a warning is preceded by **no**, it is disabled. To disable all warnings, use ''**extract warn no all**''. To see which warning options are in effect, use ''**extract warn**''.

extresist [*cell* [*threshold*]]
Postprocessor for improving on the resistance calculation performed by the circuit extractor. To use this command, you first have to extract the design rooted at *cell* with **:extract** *cell*, and then flatten the design using *ext2sim* (1), producing the files *cell***.sim** and *cell***.nodes**. Then run **:extresist** *cell* to produce a file, *cell***.res.ext**, containing differences between the network described by the **.ext** files produced the first time around, and a new network that incorporates explicit two-point resistors where appropriate (see below). This file may be appended to *cell***.ext**, and then *ext2sim*run for a second time, to produce a new network with explicit resistors. The *threshold* parameter is used to control which nodes are turned into resistor networks: any node whose total resistance exceeds *threshold* times the smallest on-resistance of any transistor connected to that node will be approximated as a resistor network.

feedback *option* [*args*]
Examine feedback information that is created by several of the Magic commands to report problems or highlight certain things for users. *Option* and *args* are used in the following ways:

feedback add *text* [*style*]
Used to create a feedback area manually at the location of the box. This is intended as a way for other programs like Crystal to highlight things on a layout. They can generate a command file consisting of a **feedback clear** command, and a sequence of **box** and

feedback add commands. *Text* is associated with the feedback (it will be printed by **feedback why** and **feedback find**). *Style* tells how to display the feedback, and is one of **dotted**, **medium**, **outline**, **pale**, and **solid** (if unspecified, *style* defaults to **pale**).

feedback clear
Clears all existing feedback information from the screen.

feedback count
Prints out a count of the current number of feedback areas.

feedback find [*nth*]
Used to locate a particular feedback area. If *nth* is specified, the box is moved to the location of the *nth* feedback area. If *nth* isn't specified, then the box is moved to the next sequential feedback area after the last one located with **feedback find**. In either event, the explanation associated with the feedback area is printed.

feedback help
Prints a short synopsis of all the **feedback** command options.

feedback save *file*
This option will save information about all existing feedback areas in *file*. The information is stored as a collection of Magic commands, so that it can be recovered with the command **source** *file*.

feedback why
Prints out the explanations associated with all feedback areas underneath the box.

fill *direction* [*layers*]
Direction is a Manhattan direction (see the section DIRECTIONS below). The paint visible under one edge of the box is sampled. Everywhere that the edge touches paint, the paint is extended in the given direction to the opposite side of the box. For example, if *direction* is **north**, then paint is sampled under the bottom edge of the box and extended to the top edge. If *layers* is specified, then only the given layers are considered; if *layers* isn't specified, then all layers are considered.

findbox [**zoom**]
Center the view on the box. If the optional **zoom** argument is present, zoom into the area specified by the box. This command will complain if the box is not in the window you are pointing to.

flush [*cellname*]
Cell *cellname* is reloaded from disk. All changes made to the cell since it was last saved are discarded. If *cellname* is not given, the edit cell is flushed.

garoute *option* [*args*]
This command, with no *option* or *arg*, is like the **route** command: it generates routing in the edit cell to make connections specified in the current netlist. (See the **route** command for further information). Unlike the **route** command, this command is intended to be used for routing types of circuits, such as gate-arrays, whose routing channels can be determined in advance, and which require the ability to river-route across the tops of cells. The channels must have been predefined using **garoute channel** commands prior to this command being invoked. Unlike the **route** command, where the box indicates the routing area, this command ignores the box entirely. The new wires are placed in the edit cell. The netlist used is that selected by the **route netlist** command, or the current netlist being edited in a **netlist** window if no **route netlist** command has been given. *Options* and *args* have the following effects:

garoute channel [*type*]

garoute channel *xlo ylo xhi yhi* [*type]*
Define a channel. If *xlo*, *ylo*, *xhi*, and *yhi* are provided, they are interpreted as the coordinates of the lower-left and upper-right of the bounding box for the channel respectively.

Otherwise, the coordinates of the box are used. The boundary of each channel is adjusted inward to lie halfway between routing grid lines if it does not lie there already; if the channel is adjusted, a warning message is printed. The channel defined is an ordinary routing channel if *type* is not specified; such channels are identical to those used by the router of the **route** command. If *type* is given, it must be either **h** or **v**. The channel thereby created will be a *river-routing* channel inside which only left-to-right routes are possible (''**h**'') or top-to-bottom (''**v**''). Unlike a normal channel, a river-routing channel may contain terminals in its interior.

garoute generate *type* [*file*]
Provides a primitive form of channel decomposition for regular structures such as gate-array or standard-cell layouts. Generates a collection of **garoute channel** commands, either to the standard output, or to *file* if the latter is specified. The *type* parameter must be either **h** or **v**. The entire area contained within the box is turned into routing channels. Each cell inside this area has its bounding box computed for purposes of routing by looking only at those layers considered to be ''obstacles'' to routing (see ''Tutorial #7: Netlists and Routing'' for details). The bounding box just computed is then extended all the way to the sides of the area of the box tool, vertically if *type* is **h** or horizontally if *type* is **v**. This extended area is then marked as belonging to a river-routing channel of type *type*; adjacent channels of this type are merged into a single channel. After all cells are processed, the areas not marked as being river-routing channels are output as normal channels.

garoute help
Print a short synopsis of all the **garoute** command options.

garoute nowarn
If a given terminal appears in more than one place inside a cell, the router can leave feedback if it is not possible to route to all of the places where the terminal appears. The **garoute nowarn** command instructs the router to leave feedback only if it is not possible to route to *any* of the locations of a terminal. (This is the default behavior of **garoute** router).

garoute route [*netlist*]
Route the edit cell. If *netlist* is not specified, the netlist used is the same as when **garoute** is given with no options. If *netlist* is given, then it is used instead.

garoute reset
Clear all channels defined by **garoute channel** in preparation for redefining a new set of channels.

garoute warn
The opposite of **garoute nowarn**, this command instructs the router to leave feedback if it is not possible to route to all of the places where a terminal appears when a terminal has more than one location, even if not all of those locations are actually selected for routing by the global router.

getcell *cellName* [**child** *refPointC*] [**parent** *refPointP*]
This command adds a child cell instance to the edit cell. The instance refers to the cell *cellName*; it is positioned so that *refPointC* in the child is at point *refPointP* in the edit cell. The reference points can either be the name of a label, in which case the lower-left corner of the label's box is used as the reference point, or as a pair of numbers giving the (*x*, *y*) coordinates of a point explicitly. If *refPointC* is not specified, the lower-left corner of *cellName* cell is used. If *refPointP* is not specified, the lower-left corner of the box tool is used (the box must be in a window on the edit cell). The new subcell is selected. The difference between this command and **dump** is that **dump** copies the contents of the cell, while **getcell** simply makes a reference to the original cell. *Cellname* must not be the edit cell or one of its ancestors.

getnode [**alias on** | **alias off**]

getnode [**abort** [*str*]]

Getnode prints out the node names (used by the extractor) for all selected paint. If aliasing turned on, getnode prints all the names it finds for a given node. It may not print every name that exists, however. When turned off, it just prints one name. The abort option allows the user to tell getnode that it is not important to completely search nodes that have certain names. For example, **getnode abort Vdd** will tell getnode not to continue searching the node if it determines that one of its names is Vdd. A **getnode abort**, without a string argument, will erase the list of names previously created by calling **getnode abort** with string arguments. Getnode can be safely aborted at any time by typing the interrupt character, usually ^C. See *Tutorial #11: Using IRSIM and RSIM with Magic* for more information on this command.

grid [*xSpacing* [*ySpacing* [*xOrigin yOrigin*]]]

grid off If no arguments are given, a one-unit grid is toggled on or off in the window underneath the cursor. **Grid off** always turns the grid off, regardless of whether it was on or off previously. If numerical arguments are given, the arguments determine the grid spacing and origin for the window under the cursor. In its most general form, **grid** takes four integer arguments. **XOrigin** and **yOrigin** specify an origin for the grid: horizontal and vertical grid lines will pass through this point. **XSpacing** and **ySpacing** determine the number of units between adjacent grid lines. If **xOrigin** and **yOrigin** are omitted, they default to 0. If **ySpacing** is also omitted, the xSpacing value is used for both spacings. Grid parameters will be retained for a window until explicitly changed by another *grid* command. When the grid is displayed, a solid box is drawn to show the origin of the edit cell.

identify *instance_id*

Set the instance identifier of the selected cell use to *instance_id*. *Instance_id* must be unique among all instance identifiers in the parent of the selected cell. Initially, Magic guarantees uniqueness of identifiers by giving each cell an initial identifier consisting of the cell definition name followed by an underscore and a small integer.

iroute *subcommand* [*args*]

This command provides an interactive interface to the Magic maze-router. Routing is done one connection at a time. Three internal *hint* layers, **magnet**, **fence**, and **rotate**, allow the user to guide routing graphically. Routes are chosen close to magnets (if possible), routing does not cross fence boundaries, and rotate areas reverse the preferred routing directions for each layer. The maze-router seeks to find a lowest-cost path. Parameters specifying costs for horizontal and vertical routing on each layer, cost for jogs and contacts, and cost (per unit area) for distance between a path and magnets, help determine the nature of the routes. Several *search* parameters permit tuning to achieve acceptable routes in as short a time as possible. Routing can always be interrupted with **^C**. The iroute subcommands are as follows:

iroute Routes from cursor to inside box.

iroute contact [*type*] [*parameter*] [*value1*] ... [*valuen*]

An asterisk, *****, can be used for *type* and *parameter*. This command is for setting and examining parameters related to contacts.

iroute help [*subcommand*]

Summarizes irouter commands. If a *subcommand* is given, usage information for that subcommand is printed.

iroute layers [*type*] [*parameter*] [*value1*] ... [*valuen*]

An asterisk, *****, can be used for *type* and *parameter*. This command is for setting and examining parameters related to route layers.

iroute route [*options*]

Invokes the router. Options are as follows:

-sLayers *layers* = layers route may start on
-sCursor = start route at cursor (DEFAULT)
-sLabel *name* = start route at label of given name
-sPoint *x y* = start route at given coordinates
-dLayers *layers* = layers route may end on
-dBox = route to box (DEFAULT)
-dLabel *name* = route to label of given name
-dRect *xbot ybot xtop ytop* = route to rectangle of given coordinates
-dSelection = *route to selection*

iroute saveParameters <*filename*>
Saves all current irouter parameter settings. The parameters can be restored to these values with the command ''**source** *filename*''.

iroute search [*searchParameter*] [*value*]
Allows parameters controlling the search to be modified. If routing is too slow try increasing **rate**. If the router is producing bad results, try reducing **rate**. Its a good idea to make **width** at least twice as big as **rate**.

iroute spacings [*route-type*] [*type*] [*spacing*] ... [*typen spacingn*]
Default minimum spacings between a route-type placed by the router and other types are derived from the **drc** section of the technology file. The defaults can be overridden by this command. The special type **SUBCELL** is used to specify minimum spacing to unexpanded subcells.

iroute verbosity [*level*]
Controls the number of messages printed during routing:
0 = errors and warnings only,
1 = brief,
2 = lots of statistics.

iroute version
Prints irouter version information.

iroute wizard [*wizardparameter*] [*value*]
Used to examine and set miscellaneous parameters. Most of these are best left alone by the unadventurous user.

label *string* [*pos* [*layer*]]
A label with text *string* is positioned at the box location. Labels may cover points, lines, or areas, and are associated with specific layers. Normally the box is collapsed to either a point or to a line (when labeling terminals on the edges of cells). Normally also, the area under the box is occupied by a single layer. If no *layer* argument is specified, then the label is attached to the layer under the box, or space if no layer covers the entire area of the box. If *layer* is specified but *layer* doesn't cover the entire area of the box, the label will be moved to another layer or space. Labels attached to space will be considered by CIF processing programs to be attached to all layers overlapping the area of the label. *Pos* is optional, and specifies where the label text is to be displayed relative to the box (e.g. ''north''). If *pos* isn't given, Magic will pick a position to ensure that the label text doesn't stick out past the edge of the cell.

layers Prints out the names of all the layers defined for the current technology.

load [*file*]
Load the cell hierarchy rooted at *file***.mag** into the window underneath the cursor. If no *file* is supplied, a new unnamed cell is created. The root cell of the hierarchy is made the edit cell unless there is already an edit cell in a different window.

move [*direction* [*amount*]]

move to *x y*

If no arguments are given, the selection is picked up by the point underneath the lower-left corner of the box and moved so that this point lies at the cursor location. If *direction* is given, it must be a Manhattan direction (e.g. **north**). The selection is moved in that direction by *amount*. If the box is in the same window as the selection, it is moved too. *Amount* defaults to 1. Selected material that is not in the edit cell, is not affected. The second form of the command is as though the cursor were pointing to (*x, y*) in the edit cell; the selection is picked up by the point beneath the lower-left corner of the box and moved so that this point lies at (*x, y*).

paint *layers*

The area underneath the box is painted in *layers*.

path [*searchpath*]

This command tells Magic where to look for cells. *Searchpath* contains a list of directories separated by colons or spaces (if spaces are used, then *searchpath* must be surrounded by quotes). When looking for a cell, Magic will check each directory in the path in order, until the cell is found. If the cell is not found anywhere in the path, Magic will look in the system library for it. If the *path* command is invoked with no arguments, the current search path is printed.

plot *option* [*args*]

Used to generate hardcopy plots direct from Magic. *Options* and *args* are used in the following ways:

plot gremlin *file* [*layers*]

Generate a Gremlin-format description of everything under the box, and write the description in *file*. If *layers* isn't specified, paint, labels, and unexpanded subcells are all included in the Gremlin file just as they appear on the screen. If *layers* is specified, then just the indicated layers are output in the Gremlin file. *Layers* may include the special layers **labels** and **subcell**. The Gremlin file is scaled to have a total size between 256 and 512 units; you should use the **width** and/or **height** Grn commands to ensure that the printed version is the size you want. Use the **mg** stipples in Grn. No plot parameters are used in Gremlin plotting.

plot help

Print a short synopsis of all the **plot** command options.

plot parameters [*name value*]

If **plot parameters** is invoked with no additional arguments, the values for all of the plot parameters are printed. If *name* and *value* are provided, then *name* is the name of a plot parameter and *value* is a new value for it. Plot parameters are used to control various aspects of plotting; all of them have ''reasonable'' initial values. Most of the parameters available now are used to control Versatec-style plotting. They are:

cellIdFont

The name of the font to use for cell instance ids in Versatec plots. This must be a file in Vfont format.

cellNameFont

The name of the font to use for cell names in Versatec plots. This must be a file in Vfont format.

color If this is set to **true**, the **:plot versatec** command will generate output suitable for a four-color Versatec plotter, using the styles defined in the **colorversatec** style of the **plot** section of the technology file. If **color** is **false** (the default), then **:plot versatec** generates normal black-and-white plots.

directory

The name of the directory in which to create raster files for the Versatec. The raster files have names of the form **magicPlotXXXXXX**, where **XXXXXX** is a

process-specific identifier.

dotsPerInch
Indicates how many dots per inch there are on the Versatec printer. This parameter is used only for computing the scale factor for plotting. Must be an integer greater than zero.

labelFont
The name of the font to use for labels in Versatec plots. This must be a file in Vfont format.

printer The name of the printer to which to spool Versatec raster files.

showcellnames
If ''true'' (the default) then the name and instance-identifier of each unexpanded subcell is displayed inside its bounding box. If this parameter is ''false'' then only the bounding box of the cell is displayed.

spoolCommand
The command used to spool Versatec raster files. This must be a text string containing two ''%s'' formatting fields. The first ''%s'' will be replaced with the printer name, and the second one will be replaced with the name of the raster file.

swathHeight
How many raster lines of Versatec output to generate in memory at one time. The raster file is generated in swaths in order to keep the memory requirements reasonable. This parameter determines the size of the swaths. It must be an integer greater than zero, and should be a multiple of 16 in order to avoid misalignment of stipple patterns.

width The number of pixels across the Versatec printer. Must be an integer greater than 0, and must be an even multiple of 32.

plot versatec [*size* [*layers*]]
Generate a raster file describing all the the information underneath the box in a format suitable for printing on Versatec black-and-white or color printers, and spool the file for printing. See the plot parameters above for information about the parameters that are used to control Versatec plotting. *Size* is used to scale the plot: a scalefactor is chosen so that the area of the box is *size* inches across on the printed page. *Size* defaults to the width of the printer. *Layers* selects which layers (including labels and subcells) to plot; it defaults to everything visible on the screen.

plow *direction* [*layers*]

plow *option* [*args*]
The first form of this command invokes the plowing operation to stretch and/or compact a cell. *Direction* is a Manhattan direction. *Layers* is an optional collection of mask layers, which defaults to *****. One of the edges of the box is treated as a plow and dragged to the opposite edge of the box (e.g. the left edge is used as the plow when **plow right** is invoked). All edges on *layers* that lie in the plow's path are pushed ahead of it, and they push other edges ahead of them to maintain design rules, connectivity, and transistor and contact sizes. Subcells are moved in their entirety without being modified internally. Any mask information overlapping a subcell moved by plowing is also moved by the same amount. *Option* and *args* are used in the following ways:

plow boundary
The box specifies the area that may be modified by plowing. This area is highlighted with a pale stipple outline. Subsequent plows are not allowed to modify any area outside that specified by the box; if they do, the distance the plow moves is reduced by an

amount sufficient to insure that no geometry outside the boundary gets affected.

plow help
Prints a short synopsis of all the **plow** command options.

plow horizon *n*

plow horizon
The first form sets the plowing jog horizon to *n* units. The second form simply prints the value of the jog horizon. Every time plowing considers introducing a jog in a piece of material, it looks up and down that piece of material for a distance equal to the jog horizon. If it finds an existing jog within this distance, it uses it. Only if no jog is found within the jog horizon does plowing introduce one of its own. A jog horizon of zero means that plowing will always introduce new jogs where needed. A jog horizon of infinity (**plow nojogs**) means that plowing will not introduce any new jogs of its own.

plow jogs
Re-enable jog insertion with a horizon of 0. This command is equivalent to **plow horizon 0**.

plow noboundary
Remove any boundary specified with a previous **plow boundary** command.

plow nojogs
Sets the jog horizon to infinity. This means that plowing will not introduce any jogs of its own; it will only use existing ones.

plow nostraighten
Don't straighten jogs automatically after each plow operation.

plow selection [*direction* [*distance*]]
Like the **move** or **stretch** commands, this moves all the material in the selection that belongs to the edit cell. However, any material not in the selection is pushed out of its way, just as though each piece of the selection were plowed individually. If no arguments are given, the selection is picked up by the point underneath the lower-left corner of the box and plowed so that this point lies at the cursor location. The box is moved along with the selection. If *direction* is given, it must be a Manhattan direction (e.g. **north**). The selection is moved in that direction by *amount*. If the box is in the same window as the selection, it is moved too. *Amount* defaults to 1. If there is selected material that isn't in the edit cell, it is ignored (note that this is different from **select** and **move**). If *direction* isn't given and the cursor isn't exactly left, right, up, or down from the box corner, then Magic first rounds the cursor position off to a position that is one of those (whichever is closest).

plow straighten
Straighten jogs automatically after each plow operation. The effect will be as though the **straighten** command were invoked after each plow operation, with the same direction, and over the area changed by plowing.

resist *cell* [*tolerance*]
This command is similar to **extresist** above, but used for extracting resistance networks for individual nodes. Only the node underneath the box is processed. The network for this node is output to the file *cell*.**res.ext**. See the description for **extresist** for an explanation of *tolerance*.

route *option* [*args*]
This command, with no *option* or *arg*, is used to generate routing using the Magic router in the edit cell to make connections specified in the current netlist. The box is used to indicate the routing area: no routing will be placed outside the area of the box. The new wires are placed in the edit cell. *Options* and *args* have the following effects:

route end [*real*]
Print the value of the channel end constant used by the channel router. If a value is supplied, the channel end constant is set to that value. The channel end constant is a dimensionless multiplier used to compute how far from the end of a channel to begin preparations to make end connections.

route help
Print a short synopsis of all the **route** command options.

route jog [*int*]
Print the value of the minimum jog length used by the channel router. If a value is supplied, the minimum jog length is set to that value. The channel router makes no vertical jogs shorter than the minimum jog length, measured in router grid units. Higher values for this constant may improve the quality of the routing by removing unnecessary jogs; however, prohibiting short jogs may make some channels unroutable.

route metal
Toggle metal maximization on or off. The route command routes the preferred routing layer (termed ''metal'') horizontally and the alternate routing layer vertically. By default wires on the alternate routing layer are then converted, as much as possible, to the preferred layer before being painted into the layout. Enabling metal maximization improves the quality of the resulting routing, since the preferred routing layer generally has better electrical characteristics; however, designers wishing to do hand routing after automatic routing may find it easier to disable metal maximization and deal with a layer-per-direction layout.

route netlist [*file*]
Print the name of the current netlist. If a file name is specified, it is opened if possible, and the new netlist is loaded. This option is provided primarily as a convenience so you need not open the netlist menu before routing.

route obstacle [*real*]
Print the obstacle constant used by the channel router. If a value is supplied, set the channel router obstacle constant to that value. The obstacle constant is a dimensionless multiplier used in deciding how far in front of an obstacle the channel router should begin jogging nets out of the way. Larger values mean that nets will jog out of the way earlier; however, if nets jog out of the way too early routing area is wasted.

route origin [*x y*]
Print the x- and y-coordinates of the origin of the routing grid. By default, the routing grid starts from (0,0). However, by supplying an *x* and *y* coordinate to the **route origin** command, the origin can be set to any other value. This command is primarily useful when routing a chip that has been designed with routing on the same pitch as the router will use, but where the left and bottom edges of the pre-existing routing don't line up with the routing grid lines (for example, the pre-existing routing might have been centered on routing grid lines). The alternative to specifying a different origin for the routing grid would be to translate all the material in the cell to be routed so that the prewiring lined up properly with routing grid lines.

route settings
Print the values of all router parameters.

route steady [*int*]
Print the value of the channel router's steady net constant. If a value is supplied, set the steady net constant to the value. The steady net constant, measured in router grid units, specifies how far beyond the next terminal the channel router should look for a conflicting terminal before deciding that a net is rising or falling. Larger values mean

that the net rises and falls less often.

route tech
Print the router technology information. This includes information such as the names of the preferred and alternate routing layers, their wire widths, the router grid spacing, and the contact size.

route viamin
Minimize vias in (previously) routed netlist. This subcommand removes unnecessary layer changes in all nets in the current netlist to minimize via count. The preferred routing layer, **layer1** in the **router** section of the technology file, is favored by the algorithm. Note that ''**route viamin**'' is an independent routing postpass that can be applied even if the routing was not generated by the **route** command, provided the layers and widths agree with the **router** section of the technology file.

route vias [*int*]
Print the value of the metal maximization via constant. If a value is supplied, set the via constant to the value. The via constant, measured in router grid units, represents the tradeoff between metal maximization and the via count. In many cases it is possible to convert wiring on the alternate routing layer into routing on the preferred routing layer (''metal'') at the expense of introducing one or two vias. The via constant specifies the amount of converted wiring that makes it worthwhile to add vias to the routing.

rsim [*options*] [*filename*]
Runs rsim under Magic. See *Tutorial #11: Using IRSIM and RSIM with Magic* for more information on what options and files are required by rsim. Normally, IRSIM requires a parameter file for the technology and a **.sim** file describing the circuit.

The **rsim** command without any options can be used to interact with a previously-started rsim. Type **rsim** and you will see the rsim prompt. To get back to magic, type **q**.

save [*name*]
Save the edit cell on disk. If the edit cell is currently the ''(UNNAMED)'' cell, *name* must be specified; in this case the edit cell is renamed to *name* as well as being saved in the file *name*.**mag**. Otherwise, *name* is optional. If specified, the edit cell is saved in the file *name*.**mag**; otherwise, it is saved in the file from which it was originally read.

see *option*
This command is used to control which layers are to be displayed in the window under the cursor. It has several forms:

see no *layers*
Do not display the given layers in the window under the cursor. If **labels** is given as a layer name, don't display labels in that window either. If **errors** is given as a layer, no design-rule violations will be displayed (the checker will continue to run, though). If *layers* is given as "*", all mask layers will be disabled, but errors and labels will still be shown. See the "LAYERS" section at the end of this manual page for an explanation of layer naming in Magic.

see *layers*
Reenable display of the given *layers*. Note that "*" expands to all mask layers, but does not include the label or error layers. See the "LAYERS" section at the end of this manual page for details.

see no Don't display any mask layers or labels. Only subcell bounding boxes will be displayed.

see Reenable display of all mask layers, labels, and errors.

see allSame
Display all cells the same way. This disables the facility where the edit cell is displayed

in bright colors and non-edit cells are in paler colors. After **see allSame**, all mask information will be displayed in bright colors.

see no allSame
Reenable the facility where non-edit cells are drawn in paler colors.

select *option*
This command is used to select paint, labels, and subcells before operating on them with commands like **move** and **copy** and **delete**. It has several forms:

select If the cursor is over empty space, then this command is identical to **select cell**. Otherwise, paint is selected. The first time the command is invoked, a chunk of paint is selected: the largest rectangular area of material of the same type visible underneath the cursor. If the command is invoked again without moving the cursor, the selection is extended to include all material of the same type, regardless of shape. If the command is invoked a third time, the selection is extended again to include all material that is visible and electrically connected to the point underneath the cursor.

select more
This command is identical to **select** except that the selection is not first cleared. The result is to add the newly-selected material to what is already in the selection.

select less
This chooses material just as **select** does, but the material is removed from the selection, rather than added to it. The result is to deselect the chosen material.

select [**more** | **less**] **area** *layers*
Select material by area. If *layers* are not specified, then all paint, labels, and unexpanded subcells visible underneath the box are selected. If *layers* is specified, then only those layers are selected. If **more** is specified, the new material is added to the current selection rather than replacing it. If **less** is specified, the new material is removed from the selection (deselected).

select [**more** | **less**] **cell** *name*
Select a subcell. If *name* isn't given, this command finds a subcell that is visible underneath the cursor and selects it. If the command is repeated without moving the cursor then it will step through all the subcells under the cursor. If *name* is given, it is treated as a hierarchical instance identifier starting from the root of the window underneath the cursor. The named cell is selected. If **more** is specified, the new subcell is added to the current selection instead of replacing it. If **less** is specified, the new subcell is removed from the selection (deselected).

select clear
Clear out the selection. This does not affect the layout; it merely deselects everything.

select help
Print a short synopsis of the selection commands.

select save *cell*
Save all the information in the selection as a Magic cell on disk. The selection will be saved in file *cell***.mag**.

select and the see command
Select interacts with the **see** command. When selecting individual pieces of material, only visible layers are candidates for selection. When selecting an entire area, however, both visible and non-visible material is selected. This behavior allows entire regions of material to be moved, even if **see** has been used to turn off the display of some of the layers.

sideways

Flip the selection left-to-right about a vertical axis running through the center of the selection's area. If the box is in the same window as the selection, it is flipped too. Selected material not in the edit cell is not affected.

simcmd *cmd*
Sends the command *cmd* to rsim for execution. See *Tutorial #11: Using IRSIM and RSIM with Magic* for more information.

snap [**on**]

snap [**off**]
Control whether the box and point are snapped to the grid selected for the windows in which they appear (the grid was set by the **grid** command), or to the standard 1x1 grid. The default is for snapping to be **off**, i.e., snapping to a 1x1 grid. With no arguments, **snap** prints whether snapping is enabled or not.

startrsim [*options*] [*filename*]
Similar to the **rsim** command, except it returns to Magic as soon as rsim is started. See *Tutorial #11: Using IRSIM and RSIM with Magic* for more information.

straighten *direction*
Straighten jogs in wires underneath the box by pulling them in *direction*. Jogs are only straightened if doing so will cause no additional geometry to move.

stretch [*direction* [*amount*]]
This command is identical to **move** except that simple stretching occurs as the selection is moved. Each piece of paint in the selection causes the area through which it's moved to be erased in that layer. Also, each piece of paint in the selection that touches unselected material along its back side causes extra material to be painted to fill in the gap left by the move. If *direction* isn't given and the cursor isn't exactly left, right, up, or down from the box corner, then Magic first rounds the cursor position off to a position that is one of those (whichever is closest).

tool [*name* | **info**]
Change the current tool. The result is that the cursor shape is different and the mouse buttons mean different things. The command **tool info** prints out the meanings of the buttons for the current tool. **Tool** *name* changes the current tool to *name*, where *name* is one of **box**, **wiring**, or **netlist**. If **tool** is invoked with no arguments, it picks a new tool in circular sequence: multiple invocations will cycle through all of the available tools.

unexpand
Unexpand all cells that touch the box but don't completely contain it.

upsidedown
Flip the selection upside down about a horizontal axis running through the center of the selection's area. If the box is in the same window as the selection then it is flipped too. Selected material that is not in the edit cell is not changed.

what Print out information about all the things that are selected.

wire *option* [*args*]
This command provides a centerline-wiring style user interface. *Option* and *args* specify a particular wiring option, as described below. Some of the options can be invoked via mouse buttons when the **wiring** tool is active.

wire help
Print out a synopsis of the various wiring commands.

wire horizontal
Just like **wire leg** except that the new segment is forced to be horizontal.

wire leg

Paint a horizontal or vertical segment of wire from one side of the box over to the cursor's x- or y-location (respectively). The direction (horizontal or vertical) is chosen so as to produce the longest possible segment. The segment is painted in the current wiring material and thickness. The new segment is selected, and the box is placed at its tip.

wire switch [*layer width***]**
Switch routing layers and place a contact at the box location. The contact type is chosen to connect the old and new routing materials. The box is placed at the position of the contact, and the contact is selected. If *layer* and *width* are specified, they are used as the new routing material and width, respectively. If they are not specified, the new material and width are chosen to correspond to the material underneath the cursor.

wire type [*layer width***]**
Pick a material and width for wiring. If *layer* and *width* are not given, then they are chosen from the material underneath the cursor, a square chunk of material is selected to indicate the layer and width that were chosen, and the box is placed over this chunk. If *layer* and *width* are given, then this command does not modify the box position.

wire vertical
Just like **wire leg** except that the new segment is forced to be vertical.

writeall [**force**]
This command steps through all the cells that have been modified in this edit session and gives you a chance to write them out. If the **force** option is specified, then ''autowrite'' mode is used: all modified cells are automatically written without asking for permission.

MOUSE BUTTONS FOR WINDOW CONTROL

For systems with pre-existing window packages, such as X windows, Magic generally uses the conventions for moving windows in those systems. For systems without pre-existing window packages, such as the AED line of displays, windows can be re-arranged by clicking mouse buttons in window borders. When pressed in the border area of a window, the left and right mouse buttons resize the window, instead of resizing the box as they would when the box tool is active. The buttons behave in the same way that they do for the box tool. For example, the left button moves the whole window by the lower left corner while the right button moves just the upper right corner.

The use of scroll bars and the middle button are explained in ''Magic Tutorial #5: Multiple Windows''.

COMMANDS FOR ALL WINDOWS

These commands are not used for layout, but are instead used for overall, housekeeping functions. They are valid in all windows.

center Adjust the view in the window under the cursor so the point underneath the cursor is at the center of the window.

closewindow
The window under the cursor is closed. That area of the screen will now show other windows or the background.

echo [**-n**] *str1 str2 ... strN*
Prints *str1 str2 ... strN* in the text window, separated by spaces and followed by a newline. If the **-n** switch is given, no newline is output after the command.

grow Grows a window up to full-screen size. Typing the command again causes the window to shrink down to its former size and position.

help [*pattern*]
Displays a synopsis of commands that apply to the window you are pointing to. If *pattern* is given then only command descriptions containing the pattern are printed. *Pattern* may contain '*' and

'?' characters, which match a string of non-blank characters or a single non-blank character (respectively).

logcommands [*file* [**update**]]]
If *file* is given, all further commands are logged to that file. If no arguments are given, command logging is terminated. If the keyword **update** is present, commands are output to the file to cause the screen to be updated after each command when the command file is read back in.

macro [*char* [*command*]]
Command is associated with *char* such that typing *char* on the keyboard is equivalent to typing ''**:**'' followed by *command*. If *command* is omitted, the current macro for *char* is printed. If *char* is also omitted, then all current macros are printed. If *command* contains spaces, tabs, or semicolons then it must be placed in quotes. The semicolon acts as a command separator allowing multiple commands to be combined in a single macro.

openwindow [*cell*]
Open a new, empty window at the cursor position. Placement, sizing, and methods of manipulation are determined by the conventions of the window system in use. If graphics is being done directly to a frame-buffer with no intervening window system, e.g. on an AED, the windows can be manipulated via mouse buttons as described in ''MOUSE BUTTONS FOR WINDOW CONTROL'' above. If *cell* is specified, then that cell is displayed in the new window. Otherwise the area of the box will be displayed in the new window.

over Move the window under the cursor so that it appears above all other windows.

pushbutton *button action*
Simulates a button push. Button should be **left**, **middle**, or **right**. Action is one of **up**, or **down**. This command is normally invoked only from command scripts produced by the **logcommands** command.

quit Exit Magic and return to the shell. If any cells, colormaps, or netlists have changed since they were last saved on disk, you are given a chance to abort the command and continue in Magic.

redo [*n*]
Redo the last *n* commands that were undone using **undo** (see below). The number of commands to redo defaults to 1 if *n* is not specified.

redraw Redraw the graphics screen.

reset Reset the graphics controller and redraw the graphics screen. You should usually reset the graphics hardware manually before invoking this command. This command is a way to recover from noise errors on serial lines, and thus is ignored on workstations with built-in frame buffers.

scroll *direction* [*amount*]
The window under the cursor is moved by *amount* screenfulls in *direction* relative to the circuit. If *amount* is omitted, it defaults to 0.5.

send *type command*
Send a *command* to the window client named by *type*. The result is just as if *command* had been typed in a window of type *type*. See **specialopen**, below, for the allowable types of windows.

setpoint [*x y* [*windowID*]]
Fakes the location of the cursor up until after the next interactive command. Without arguments, just prints out the current point location. This command is normally invoked only from command scripts produced by the **logcommands** command or by wizards that are using Magic without a color display.

If *windowID* is given, then the point is assumed to be in that window's screen coordinate system rather than absolute screen coordinates. This feature is needed for devices like the Sun 160 that have separate coordinate systems for each window. To find out a window's ID on such a device,

turn on command logging and look at the file produced.

sleep *n* Causes Magic to go to sleep for *n* seconds.

source *filename*
Each line of *filename* is read and processed as one command. No colons are necessary. Any line whose last character is backslash is joined to the following line. The commands **setpoint**, **pushbutton**, **echo**, **sleep**, and **updatedisplay** are useful in command files, and seldom used elsewhere.

specialopen [*x1 y1 x2 y2*] *type* [*args*]
Open a window of type *type*. If the optional *x1 y1 x2 y2* coordinates are given, then the new window will have its lower left corner at screen coordinates (*x1*, *y1*) and its upper right corner at screen coordinates (*x2*, *y2*). The *args* arguments are interpreted differently depending upon the type of the window. These types are known:

layout This type of window is used to edit a VLSI cell. The command takes a single argument which is used as the name of a cell to be loaded. The command

open *filename*

is a shorthand for the command

specialopen layout *filename*.

color This type of window allows the color map to be edited. See the section COMMANDS FOR COLORMAP EDITING below.

netlist This type of window presents a menu that can be used to place labels, and to generate and edit net-lists. See the section COMMANDS FOR NETLIST EDITING below.

underneath
Move the window pointed at so that it lies underneath the rest of the windows.

undo [*count*]
Undoes the last *count* commands. Almost all commands in Magic are now undo-able. The only holdouts left are cell expansion/unexpansion, and window modifications (change of size, zooming, etc.). If *count* is unspecified, it defaults to 1. Only the last twenty modifications are recorded for undoing.

updatedisplay
Update the display. This command is normally invoked only from command scripts produced by the **logcommands** command. Command scripts that do not contain this command update the screen only at the end of the script.

view Choose a view for the window underneath the cursor so that everything in the window is visible.

windscrollbars [*on*|*off*]
Set the flag that determines if new windows will have scroll bars.

windowpositions [*file*]
Write out the positions of the windows in a format suitable for the **source** command. If *file* is specified, then write it out to that file instead of to the terminal.

zoom [*factor*]
Zoom the view in the window underneath the cursor by *factor*. If *factor* is less than 1, we zoom in; if it is greater than one, we zoom out.

MOUSE BUTTONS FOR NETLIST WINDOWS

When the netlist menu is opened using the command **special netlist**, a menu appears on the screen. The colored areas on the menu can be clicked with various mouse buttons to perform various actions, such as placing labels and editing netlists. For details on how to use the menu, see ''Magic Tutorial #7: Netlists and Routing''. The menu buttons all correspond to commands that could be typed in netlist or layout windows.

COMMANDS FOR NETLIST WINDOWS

The commands described below work if you are pointing to the interior of the netlist menu. They may also be invoked when you are pointing at another window by using the **send netlist** command. Terminal names in all of the commands below are hierarchical names consisting of zero or more cell use ids separated by slashes, followed by the label name, e.g. **toplatch/shiftcell_1/in**. When processing the terminal paths, the search always starts in the edit cell.

add *term1 term2*

Add the terminal named *term1* to the net containing terminal *term2*. If *term2* isn't in a net yet, make a new net containing just *term1* and *term2*.

cleanup

Check the netlist to make sure that for every terminal named in the list there is at least one label in the design. Also check to make sure that every net contains at least two distinct terminals, or one terminal with several labels by the same name. When errors are found, give the user an opportunity to delete offending terminals and nets. This command can also be invoked by clicking the ''Cleanup'' menu button.

cull Examine the current netlist and the routing in the edit cell, and remove those nets from the netlist that are already routed. This command is often used after pre-routing nets by hand, so the router won't try to implement them again.

dnet *name name ...*

For each *name* given, delete the net containing that terminal. If no *name* is given, delete the currently-selected net, just as happens when the ''No Net'' menu button is clicked.

dterm *name name ...*

For each *name* given, delete that terminal from its net.

extract Pick a piece of paint in the edit cell that lies under the box. Starting from this, trace out all the electrically-connected material in the edit cell. Where this material touches subcells, find any terminals in the subcells and make a new net containing those terminals. Note: this is a different command from the **extract** command in layout windows.

find *pattern* [*layers*]

Search the area beneath the box for labels matching *pattern*, which may contain the regular-expression characters ''*****'' ''**?**'', ''**[**'', ''**]**'', and ''\'' (as matched by *csh* (1); see the description of the **find** button in ''Magic Tutorial #7: Netlists and Routing''). For each label found, leave feedback whose text is the layer on which the label appears, followed by a semicolon, followed by the full hierarchical pathname of the label. The feedback surrounds the area of the label by one unit on all sides. (The reason for the one-unit extension is that feedback rectangles must have positive area, while labels may have zero width or height). If *layers* are given, only labels attached to those layers are considered.

flush [*netlist*]

The netlist named *netlist* is reloaded from the disk file *netlist*.**net**. Any changes made to the netlist since the last time it was written are discarded. If *netlist* isn't given, the current netlist is flushed.

join *term1 term2*

Join together the nets containing terminals *term1* and *term2*. The result is a single net containing all the terminals from both the old nets.

netlist [*name*]

Select a netlist to work on. If *name* is provided, read *name*.**net** (if it hasn't already been read before) and make it the current netlist. If *name* isn't provided, use the name of the edit cell instead.

print [*name*]

Print the names of all the terminals in the net containing *name*. If *name* isn't provided, print the

terminals in the current net. This command has the same effect as clicking on the ''Print'' menu button.

ripup [**netlist**]
This command has two forms. If **netlist** isn't typed as an argument, then find a piece of paint in the edit cell under the box. Trace out all paint in the edit cell that is electrically connected to the starting piece, and delete all of this paint. If **netlist** is typed, find all paint in the edit cell that is electrically connected to any of the terminals in the current netlist, and delete all of this paint.

savenetlist [*file*]
Save the current netlist on disk. If *file* is given, write the netlist in *file*.**net**. Otherwise, write the netlist back to the place from which it was read.

shownet
Find a piece of paint in any cell underneath the box. Starting from this paint, trace out all paint in all cells that is electrically connected to the starting piece and highlight this paint on the screen. To make the highlights go away, invoke the command with the box over empty space. This command has the same effect as clicking on the ''Show'' menu button.

showterms
Find the labels corresponding to each of the terminals in the current netlist, and generate a feedback area over each. This command has the same effect as clicking on the ''Terms'' menu button.

trace [*name*]
This command is similar to **shownet** except that instead of starting from a piece of paint under the box, it starts from each of the terminals in the net containing *name* (or the current net if no *name* is given). All connected paint in all cells is highlighted.

verify Compare the current netlist against the wiring in the edit cell to make sure that the nets are implemented exactly as specified in the netlist. If there are discrepancies, feedback areas are created to describe them. This command can also be invoked by clicking the ''Verify'' menu button.

writeall Scan through all the netlists that have been read during this editing session. If any have been modified, ask the user whether or not to write them out.

MOUSE BUTTONS FOR COLORMAP WINDOWS

Color windows display two sets of colored bars and a swatch of the color being edited. The left set of color bars is labeled Red, Green, and Blue; these correspond to the proportion of red, green, and blue in the color being edited. The right set of bars is labeled Hue, Saturation, and Value; these correspond to the same color but in a space whose axes are hue (spectral color), saturation (spectral purity vs. dilution with white), and value (light vs. dark).

The value of a color is changed by pointing inside the region spanned by one of the color bars and clicking any mouse button. The color bar will change so that it extends to the point selected by the crosshair when the button was pressed. The color can also be changed by clicking a button over one of the ''pumps'' next to a color bar. A left-button click makes a 1% increment or decrement, and a right-button click makes a 5% change.

The color being edited can be changed by pressing the left button over the current color box in the editing window, then moving the mouse and releasing the button over a point on the screen that contains the color to be edited. A color value can be copied from an existing color to the current color by pressing the right mouse button over the current color box, then releasing the button when the cursor is over the color whose value is to be copied into the current color.

COMMANDS FOR COLORMAP WINDOWS

These commands work if you are pointing to the interior of a colormap window. The commands are:

color [*number*]
Load *number* as the color being edited in the window. *Number* must be an octal number between 0 and 377; it corresponds to the entry in the color map that is to be edited. If no *number* is given, this command prints out the value of the color currently being edited.

load [*techStyle displayStyle monitorType*]
Load a new color map. If no arguments are specified, the color map for the current technology style (e.g, **mos**), display style (e.g, **7bit**), and monitor type (e.g, **std**) is re-loaded. Otherwise, the color map is read from the file *techStyle*.*displayStyle*.*monitorType***.cmap** in the current directory or in the system library directory.

save [*techStyle displayStyle monitorType*]
Save the current color map. If no arguments are specified, save the color map in a file determined by the current technology style, display style, and monitor type as above. Otherwise, save it in the file *techStyle*.*displayStyle*.*monitorType***.cmap** in the current directory or in the system library directory.

DIRECTIONS

Many of the commands take a direction as an argument. The valid direction names are **north**, **south**, **east**, **west**, **top**, **bottom**, **up**, **down**, **left**, **right**, **northeast**, **ne**, **southeast**, **se**, **northwest**, **nw**, **southwest**, **sw**, and **center**. In some cases, only Manhattan directions are permitted, which means only **north**, **south**, **east**, **west**, and their synonyms, are allowed.

LAYERS

The mask layers are different for each technology, and are described in the technology manuals. The layers below are defined in all technologies:

***** All mask layers. Does not include special layers like the label layer and the error layer (see below).

$ All layers underneath the cursor.

errors Design-rule violations (useful primarily in the **see** command).

labels Label layer.

subcell Subcell layer.

Layer masks may be formed by constructing comma-separated lists of individual layer names. The individual layer names may be abbreviated, as long as the abbreviations are unique. For example, to indicate polysilicon and n-diffusion, use **poly,ndiff** or **ndiff,poly**. The special character **–** causes all subsequent layers to be subtracted from the layer mask. For example, ***–p** means ''all layers but polysilicon''. The special character **+** reverses the effect of a previous **–**; all subsequent layers are once again added to the layer mask.

SEE ALSO

magicusage(1), ext2sim(1), sleeper(1), fsleeper(1), rsleeper(1), cmap(5), dstyle(5), ext(5), glyphs(5), magic(5), displays(5), net(5)

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

''Magic Technology Manual #2: SCMOS''
etc.

''Magic Maintainer's Manual #1: Hints for System Maintainers''
etc.

FILES

~cad/lib/magic/sys/.magic	startup file to create default macros
~/.magic	user-specific startup command file
~cad/lib/magic/nmos/*	some standard nmos cells
~cad/lib/magic/scmos/*	some standard scmos cells
~cad/lib/magic/sys/*.cmap*	colormap files, see CMAP(5) man page
~cad/lib/magic/sys/*.dstyle*	display style files, see DSTYLE(5) man page
~cad/lib/magic/sys/*.glyphs	cursor and window bitmap files, see GLYPH(5) man page
~cad/lib/magic/sys/*.tech*	technology files, see ''Maintainer's Manual #2: The Technology File''
~cad/lib/displays	configuration file for Magic serial-line displays

CAD_HOME variable. If the shell environment variable **CAD_HOME** is set, Magic uses that location instead of the true ~cad location whenever it sees a file name beginning with ~cad. This allows Magic to be run without creating an actual user called "cad".

Search path. Magic's system and library files, such as technology files and display-style files, normally are placed in the ~cad/lib/magic area. However, Magic first tries to find them in the user's current directory. This makes it easier for an individual user to override installed system files.

AUTHORS

Original: Gordon Hamachi, Robert Mayo, John Ousterhout, Walter Scott, George Taylor

Contributors: Michael Arnold (Magic maze-router and Irouter command), Don Stark (new contact scheme, X11 interface, various other things), Mike Chow (Rsim interface). The X11 driver is the work of several people, including Don Stark, Walter Scott, and Doug Pan. Many other people have contributed to Magic, but it is impossible to list them all here. We appreciate their help!

BUGS

If Magic gets stuck for some reason, try using 'kill -TERM' on it to save your cells in '*cell***.save.mag**'.

Report bugs to **magic@ucbarpa.Berkeley.EDU**. Please be specific: tell us exactly what you did to cause the problem, what you expected to happen, and what happened instead. If possible send along small files that we can use to reproduce the bug. A list of known bugs and fixes is also available from the above address.

Magic will not run under the Bourne shell (but we don't know why).

NAME

magicusage – print the names of all cells and files used in a Magic design

SYNOPSIS

magicusage [**–T** *technology*] [**–p** *path*] *rootcell*

DESCRIPTION

Magicusage will print the names of all cells and files used in the design whose root cell is *rootcell*. Each line of the output is of the form

cellname **:::** *filename*

where *cellname* is the name of the cell as it is used, and *filename* is the **.mag** file containing the cell. If a cell is not found, a line of the form

cellname **:::** << **not found** >>

is output instead.

If **–p** *path* is specified, the search path used to find **.mag** files will be *path*. Otherwise, the search path is initialized by first reading the system-wide .magic file in ~cad/lib/magic/sys, then the .magic file in the user's home directory, and finally the .magic file in the current directory. The most recent **path** command read from the three files determines the search path used to find cells.

In addition, a library path of ~cad/lib/magic/*techname* is used when searching for cells. By default, *techname* is the technology of the first cell read, but it may be overridden by specifying an explicit technology with the **–T** *techname* flag.

FILES

~cad/lib/magic/*tech*
~cad/lib/magic/sys/.magic
~/.magic

SEE ALSO

magic (1), magic (5)

AUTHOR

Walter Scott

NAME

net2ir – produce :iroute commands to route a netlist composed of two-point nets

SYNOPSIS

net2ir *feedfile netfile*

DESCRIPTION

Net2ir is used to produce commands for the Magic interactive hint router to route the collection of two-point nets specified in the *net* (5) file *netfile*, in the order in which they appear in the file. The label locations come from *feedfile*, which should consist of a series of **box** and **feedback add** Magic commands, such as produced by the **find** command (in a Magic netlist window). The text associated with each feedback command must be of the form *layer***;***label*, where *layer* is the Magic layer on which *label* lies.

The output of *net2ir* is a sequence of **:iroute route** commands, one for each net in the netlist file.

SEE ALSO

magic (1), net (5)

AUTHOR

Walter Scott

NAME

rsleeper – run sleeper remotely

SYNOPSIS

rsleeper *remotemachine*

DESCRIPTION

Rsleeper is used if you wish to run a program such as *magic* (1) on a different machine (*remotemachine*) than the one to which a graphics terminal is attached, and the local graphics terminal has a login process.

To use it, log in on the graphics terminal and run *rsleeper*. The tty printed will be on the remote machine, and can be used as the graphics display device for programs such as *magic* (1).

For *rsleeper* to work, there must be an account **sleeper** on the remote machine. Its login shell should be the program *sleeper* (1). Users must be able to rlogin to the **sleeper** account without supplying a password.

SEE ALSO

fsleeper (1), magic (1), sleeper (1), displays (5)

NAME

sim2spice – convert from .sim format to spice format

SYNOPSIS

sim2spice [–d defs] file.sim

DESCRIPTION

Sim2spice reads a file in **.sim** format and creates a new file in spice format. The file contains just a list of transistors and capacitors, the user must add the transistor models and simulation information. The new file is appended with the tag **.spice**. One other file is created, which is a list of **.sim** node names and their corresponding spice node numbers. This file is tagged **.names**.

Defs is a file of definitions. A definition can be used to set up equivelences between **.sim** node names and spice node numbers. The form of this type of definition is:

set *sim_name spice_number* [*tech*]

The *tech* field is optional. In NMOS, a special node, 'BULK', is used to represent the substrate node. For CMOS, two special nodes, 'NMOS' and 'PMOS', represent the substrate nodes for the 'n' and 'p' transistors, repectively. For example, for NMOS the **.sim** node 'GND' corresponds to spice node 0, 'Vdd' corresponds to spice node 1, and 'BULK' corresponds to spice node 2. The *defs* file for this set up would look like this:

```
set GND 0 nmos
set Vdd 2 nmos
set BULK 3 nmos
```

A definition also allows you to set a correspondence between **.sim** transistor types and and spice transistor types. The form of this definition is:

def *sim_trans spice_trans* [*tech*]

Again, the *tech* field is optional. For NMOS these definitions would look as follows:

```
def e ENMOS nmos
def d DNMOS nmos
```

Definitions may also be placed in the '.cadrc' file, but the definitions in the *defs* file overrides those in the '.cadrc' file.

SEE ALSO

ext2sim(1), magic(1), spice(1), cadrc(5), ext(5), sim(5)

AUTHOR

Dan Fitzpatrick CMOS fixes by Neil Soiffer

BUGS

The only pre-defined technologies are **nmos**, **cmos-pw**, and **cmos** (the same as **cmos-pw**). Only one definition file is allowed.

NAME

sleeper – acquire a graphics terminal and hang around

SYNOPSIS

sleeper

DESCRIPTION

Certain programs such as *magic* (1) can require the use of a graphics terminal separate from the terminal used to run the program. If the graphics terminal has an ordinary login process running on it, it is necessary to run *sleeper* to acquire ownership of the terminal, set up its modes appropriately, and prevent the login process from eating input destined for the CAD tool.

When *sleeper* is run, it will print a message of the form:

tty is:
/dev/tty*name*

Here, **/dev/tty***name* is the device name of the graphics terminal. This is particularly useful when sleeper is run over the network, or when using *fsleeper* (1) or *rsleeper* (1).

Sleeper may be killed by sending it two **QUIT** signals within ten seconds of each other. This is most easily done by typing two quit characters (usually CTRL-\ or CTRL-SHIFT-L) in a row on the graphics terminal.

For *sleeper* to work best, there should be an account named **sleeper**, whose login shell is ~cad/bin/sleeper and with no password. This enables users to log in as the user **sleeper**, and is also necessary for the programs *fsleeper* (1) and *rsleeper* (1) to work. (Note that you will have to include the full pathname of ~cad/bin/sleeper in /etc/passwd; the initial ~cad does not get expanded).

SEE ALSO

fsleeper(1), magic(1), rsleeper(1)

NAME

dqueue – procedures for managing double-ended queues in libmagicutils.a

SYNOPSIS

```
#include magic.h
#include malloc.h
#include dqueue.h

DQInit(q, capacity)
DQueue *q;
int capacity;

DQFree(q)
DQueue *q;

DQPushFront(q, elem)
DQueue *q;
ClientData elem;

DQPushRear(q, elem)
DQueue *q;
ClientData elem;

ClientData DQPopFront(q)
DQueue *q;

ClientData DQPopRear(q)
DQueue *q;

DQChangeSize(q, newSize)
DQueue *q;
int newSize

DQCopy(dst, src)
DQueue *dst;
DQueue *src;

bool DQIsEmpty(q)
DQueue *q;
```

DESCRIPTION

These procedures manipulate double-ended queues. A double-ended queue (*DQueue*) is a structure to which single word elements of type *ClientData* (actually type *(char *)*, but intended to mean ''any one-word type at all'') may be added to either end or removed from either end. Callers should not reference fields of a *DQueue* directly, but rather should use the following procedures:

DQInit initializes the DQueue *q* to have sufficient capacity to hold *capacity* entries at first. If more than this many entries are pushed on the queue, it automatically doubles its size (at the cost of copying, however), so *capacity* should be treated as the expected number of entries on the queue rather than the maximum number. *DQFree* frees the storage allocated by *DQinit* for the DQueue *q*.

DQPushFront and *DQPushRear* each place a new entry *elem* on the queue *q*; *DQPushFront* places it on the front of the queue, while *DQPushRear* places it on the rear. If the current maximum size of the queue would be exceeded by either operation, twice as much space is allocated automatically and the existing queue contents are copied to the bigger area. *DQPopFront* and *DQPopRear* remove an element from respectively the front or rear of the DQueue *q* and return it. If no elements are left, they return *NULL* (zero).

Although *DQPushFront* and *DQPushRear* take care of increasing the space for a queue automatically, sometimes it is desirable to change the size of a queue explicitly. This can be done with *DQChangeSize*, which changes the size of the DQueue *q* to *newSize*, as long as *newSize* is at least as great as the number of entries already in the queue. If there are more than *newSize* entries in the queue, nothing happens.

One DQueue may be copied to another by *DQCopy*, which copies the DQueue *src* to the DQueue *dst*.

Finally, to check whether a queue *q* is empty, one may call *DQIsEmpty*, which returns **TRUE** (non-zero) if the queue is empty, or **FALSE** (zero) if it contains any elements.

BUGS

Using **NULL** to indicate end-of-queue in *DQPopFront* and *DQPopRear* is of marginal usefulness. Callers should stick to using *DQIsEmpty* unless they are certain not to have pushed any zero elements on the queue.

SEE ALSO

magicutils (3)

NAME

extflat – procedures in libextflat.a for flattening extractor **.ext** files

SYNOPSIS

```
#include "hash.h"
#include "extflat.h"

typedef struct hiername { ... } HierName;
typedef struct efnn { ... } EFNodeName;
typedef struct efnhdr { ... } EFNodeHdr;
typedef struct efnode { ... } EFNode;
typedef struct fet { ... } Fet;


EFInit()

EFDone()

char *
EFArgs(argc, argv, argsProc, cdata)
   int argc;
   char *argv[];
   Void (*argsProc)(pargc, pargv, cdata);
   ClientData cdata;

EFReadFile(name)
   char *name;

EFFlatBuild(rootName, flags)
   char *rootName;
   int flags;

EFFlatDone()

EFVisitCaps(capProc, cdata)
   int (*capProc)(hn1, hn2, double cap, cdata);
   ClientData cdata;

EFVisitFets(fetProc, cdata)
   int (*fetProc)(fet, prefix, trans, cdata)
   ClientData cdata;

EFVisitNodes(nodeProc, cdata)
   int (*nodeProc)(node, int r, double c, cdata);
   ClientData cdata;

int
EFNodeResist(node)
   EFNode *node;

EFVisitResists(resProc, cdata)
   int (*resProc)(hn1, hn2, res, cdata);
```

```
    ClientData cdata;

bool
EFLookDist(hn1, hn2, pMinDist, pMaxDist)
    HierName *hn1, *hn2;
    int *pMinDist, *pMaxDist;

char *
EFHNToStr(hn)
    HierName *hn;

HierName *
EFStrToHN(prefix, suffixStr)
    HierName *prefix;
    char *suffixStr;

HierName *
EFHNConcat(prefix, suffix)
    HierName *prefix, *suffix;

HashEntry *
EFHNLook(prefix, suffixStr, errorStr)
    HierName *prefix;
    char *suffixStr;
    char *errorStr;

HashEntry *
EFHNConcatLook(prefix, suffix, errorStr)
    HierName *prefix, *suffix;
    char *errorStr;

EFHNOut(hn, outf)
    HierName *hn;
    FILE *outf;

EFHNFree(hn, prefix, type)
    HierName *hn, *prefix;
    int type;

bool
EFHNBest(hn1, hn2)
    HierName *hn1, *hn2;

bool
EFHNIsGND(hn)
    HierName *hn;

bool
EFHNIsGlob(hn)
    HierName *hn;

typedef struct hiername { ... } HierName;
```

typedef struct efnn { ... } EFNodeName;
typedef struct efnhdr { ... } EFNodeHdr;
typedef struct efnode { ... } EFNode;
typedef struct fet { ... } Fet;

DESCRIPTION

This module provides procedures for reading, flattening, and traversing the hierarchical extracted circuits (in *ext* (5) format) produced by the Magic circuit extractor.

To use the procedures in this library, a client should first call *EFInit* to initialize various hash tables. When a client is finally finished with this library, and wishes to free up any remaining memory used by it, *EFDone* should be called.

COMMAND-LINE ARGUMENT PROCESSING

The procedure *EFArgs* is provided for parsing of command-line flags; it should be passed the arguments to *main*. It will scan through them, recognizing those specific to *extflat* (see *extcheck* (1) for a list of these arguments) and passing unrecognized arguments to the user-supplied procedure *argsProc*, which should update **pargc* and **pargv* to point after each argument it recognizes, or else print an error message if the argument is unrecognized. If it is necessary to pass any additional information to *argsProc*, the *cdata* argument of *EFArgs* is automatically passed as the third argument to *argsProc*. If *argsProc* is NULL, any arguments not recognized by *EFArgs* are considered to be errors. *EFArgs* considers any argument not beginning with a dash (''–'') to be a filename, of which there can be at most one. The argument containing this filename is returned to the caller.

FLATTENING A CIRCUIT

Once command-line argument processing is complete, the caller can cause *ext* (5) files to be read by calling *EFReadFile*. This procedure will read *name***.ext** and all of the **.ext** files it refers to, recursively until the entire tree rooted at *name* has been read and converted into an internal, hierarchical representation. *EFReadFile* may be called several times with different values of *name*; any portions of the tree rooted at *name* that aren't already read in will be.

To build up the flat representation of a circuit read using *EFReadFile* one should call *EFFlatBuild*. The argument *rootName* gives the name of the cell, which should have been read with *EFReadFile* above, that is the root of the hierarchical circuit to be flattened. After all subsequent processing of the flat design is complete, the caller may call *EFFlatDone* to free the memory associated with the flattened circuit, possibly in preparation for calling *EFFlatBuild* with a different *rootName*.

A different procedure is provided for visiting all of the structures of each type in the flattened circuit: *EFVisitCaps*, *EFVisitFets*, *EFVisitNodes*, and *EFVisitResists*. Each takes two arguments: a search procedure to apply to all structures visited, and a ClientData field used to pass additional information to this search procedure.

EFVisitCaps visits each of the internodal capacitors in the flat circuit, applying *capProc* to each. The arguments to *capProc* are the HierNames *hn1* and *hn2* of the two nodes between which the capacitor sits, the capacitance *cap* in attofarads (type double from 6.5 and later), and the client data *cdata* with which *EFVisitCaps* was called. If it's necessary to obtain a pointer to the flat EFNode structures to which *hn1* or *hn2* refer, they can be passed to *EFHNLook* (see below).

EFVisitFets visits each of the transistors in the circuit, applying *fetProc* to each. The arguments to *fetProc* are the transistor structure itself, *fet*, the hierarchical path *prefix* that should be prepended to the node names of all the fet's terminals, a geometric transform that must be applied to all coordinates in the fet to convert them to root coordinates, the computed length *l* and width *w* of the transistor channel (taking into account substitution of symbolic values with the **–s** flag), and the client data *cdata* with which *EFVisitFets*

was called.

EFVisitNodes visits each of the flat nodes in the circuit, applying *nodeProc* to each. The arguments to *nodeProc* are the flat EFNode *node*, its lumped resistance *r* and capacitance to substrate *c* (r type is integer and c type is double from 6.5 and later), and the client data *cdata* with which *EFVisitNodes* was called. An auxiliary procedure, *EFNodeResist*, is provided to compute the lumped resistance of a node from the perimeter and area information stored in it; it returns the resistance estimate in milliohms.

EFVisitResists visits each of the explicit resistors in the circuit, applying *resProc* to each. The arguments to *resProc* are similar to those of *capProc*: the HierNames *hn1* and *hn2* of the two terminals of the resistor, its resistance *res*, and the client data *cdata* with which *EFVisitResists* was called.

A final procedure is provided for looking up distance information. *EFLookDist* searches to find if there was a distance measured between the points with the HierNames *hn1* and *hn2*. If there was a distance found, it returns TRUE and leaves **pMinDist* and **pMaxDist* set respectively to the minimum and maximum measured distance between the two points; otherwise, it returns FALSE.

NODE ORGANIZATION

Each electrical node in the circuit is represented by an *EFNode* structure, which points to a NULL-terminated list of *EFNodeName*s, each of which in turn points to the *HierName* list representing the hierarchical name. *EFNode*s contain capacitance, perimeter, and area information for a node. If this information is not required, an application may use *EFNodeHdr* structures in place of *EFNode*s in many cases; an *EFNodeHdr* consists of just the first few fields of an *EFNode*. Each *EFNodeName* is pointed to by a *HashEntry* in a hash table of all flattened node names.

HIERARCHICAL NAME MANIPULATION

Hierarchical node names are represented as lists of *HierName* structures. These structures store a hierarchical pathname such as **foo/bar[1][3]/bletch** in reverse order, with the last component (*e.g.*, **bletch**) first. Pathnames sharing a common prefix can therefore be shared.

EFStrToHN is the fundamental procedure for creating HierNames; it builds a path of HierNames from the string *suffixStr*, and then leaves this path pointing to the prefix path *prefix*. For example, if *prefix* were the path of HierNames representing **foo/bar[1][3]**, and *suffix* were the string **shift/Vb1**, the resulting HierName would be **foo/bar[1][3]/shift/Vb1**, but only the **shift/Vb1** part would be newly allocated. *EFHNFree* frees the memory allocated for the portions of the HierName path pointed to by *hn* between *hn* and *prefix*, which should be the same as the *prefix* passed to *EFStrToHN*. The *type* parameter is used only for measuring memory usage and should be zero. *EFHNToStr* converts a HierName back into a string; it returns a pointer to a statically-allocated copy of the string representation of the HierName *hn*.

EFHNConcat is like *EFStrToHN* in that it concatenates a prefix and a suffix, but the suffix passed to *EFHNConcat* has already been converted to a HierName. *EFHNConcat* creates a copy of the HierName path *suffix* whose final element points to the prefix *prefix*, in effect producing the concatenation of the two HierNames.

EFHNLook finds the HashEntry in the flat node hash table corresponding to the HierName that is the concatenation of the HierName *prefix* and the HierName formed from the suffix string *suffixStr*. The value field of this HashEntry (obtained through *HashGetValue*) is a pointer to an EFNodeName, which in turn points to the EFNode for this name. *EFHNLook* returns NULL if there wasn't an entry in the node hash table by this name, and also prints an error message of the form ''*errorStr*: node *prefix/suffixStr* not found''. *EFHNConcatLook* performs a similar function, but its second argument is a HierName instead of a string.

EFHNOut writes the HierName *hn* to the output FILE **outf*. The **-t** flag can be passed to *EFArgs* to request suppression of trailing ''**!**'' or ''**#**'' characters in node names when they are output by *EFHNOut*.

Three predicates are defined for HierNames. *EFHNBest* returns TRUE if *hn1* is ''preferred'' to *hn2*, or FALSE if the opposite is true. Global names (ending in ''**!**'') are preferred to ordinary names, which are preferred to automatically-generated names (ending in ''**#**''). Among two names of the same type, the one with the least number of pathname components is preferred. If two names have the same number of components, the one lexicographically earliest is preferable. *EFHNIsGND* returns TRUE if its argument is the ground node ''**GND!**''. *EFHNIsGlob* returns TRUE if its argument is a global node name, i.e., ends in an exclamation point.

SEE ALSO

extcheck (1), ext2dlys (1), ext2sim (1), ext2spice (1), magic (1) magicutils (3), ext (5)

NAME

geometry – primitive geometric structures and procedures in libmagicutils.a

SYNOPSIS

```
#include geometry.h

typedef struct { int p_x, p_y;

typedef struct { Point r_ll, r_ur;
#define r_xbot r_ll.p_x
#define r_ybot r_ll.p_y
#define r_xtop r_ur.p_x
#define r_ytop r_ur.p_y

typedef struct G1 { Rect r_r;

typedef struct { int t_a, t_b,

#define GEO_CENTER  0
#define GEO_NORTH   1
#define GEO_NORTHEAST2
#define GEO_EAST    3
#define GEO_SOUTHEAST4
#define GEO_SOUTH   5
#define GEO_SOUTHWEST6
#define GEO_WEST    7
#define GEO_NORTHWEST8

bool GEO_OVERLAP(r1, r2)
Rect *r1;
Rect *r2;

bool GEO_TOUCH(r1, r2)
Rect *r1;
Rect *r2;

bool GEO_SURROUND(r1, r2)
Rect *r1;
Rect *r2;

bool GEO_SURROUND_STRONG(r1, r2)
Rect *r1;
Rect *r2;

bool GEO_ENCLOSE(p, r)
Point *p;
Rect *r;

bool GEO_RECTNULL(r)
Rect *r;

GEO_EXPAND(src, amount, dst)
Rect *src, *dst;
int amount;

Transform GeoIdentityTransform;
Transform GeoUpsideDownTransform;
Transform GeoSidewaysTransform;
Transform Geo90Transform;
Transform Geo180Transform;
```

```
Transform Geo270Transform;

Rect GeoNullRect;

GeoTransPoint(t, psrc, pdst)
Transform *t;
Point *psrc, *pdst;

GeoTransRect(t, rsrc, rdst)
Transform *t;
Rect *rsrc, *rdst;

GeoTranslateTrans(tsrc, x, y, tdst)
Transform *tsrc, *tdst;
int x, y;

GeoTransTranslate(x, y, tsrc, tdst)
Transform *tsrc, *tdst;
int x, y;

GeoTransTrans(t1, t2, tdst)
Transform *t1, *t2, *tdst;

GeoInvertTrans(tsrc, tinv)
Transform *tsrc, *tinv;

int GeoScale(t)
Transform *t;

GeoDecomposeTransform(t, upsidedown, angle)
Transform *t;
bool *upsidedown;
int *angle;

int GeoNameToPos(name, manhattan, printerrors)
char *name;
bool manhattan, printerrors;

char *GeoPosToName(pos)
int pos;

int GeoTransPos(t, pos);
Transform *t;
int pos;

bool GeoInclude(src, dst);
Rect *src, *dst;

bool GeoIncludeAll(src, dst);
Rect *src, *dst;

bool GeoIncludePoint(src, dst);
Point *src;
Rect *dst;

GeoClip(r, cliparea)
Rect *r, *cliparea;

GeoClipPoint(p, cliparea)
Point *p;
Rect *cliparea;
```

bool GeoDisjoint(area, cliparea, func, cdata)
Rect *area, *cliparea;
bool (*func)(rect, cdata);
ClientData cdata;

bool GeoDummyFunc(rect, cdata)
Rect *rect;
ClientData cdata;

GeoCanonicalRect(rsrc, rdst)
Rect *rsrc, *rdst;

int GeoRectPointSide(r, p)
Rect *r;
Point *p;

int GeoRectRectSide(r1, r2)
Rect *r1, *r2;

bool GetRect(f, nskip, r)
FILE *f;
int nskip;
Rect *r;

DESCRIPTION

These procedures implement a number of useful geometric primitives: a *Point*, which consists of an integer *x* and *y* coordinate, and a *Rect*, which describes a rectangle by its lower-left and upper-right *Point*s. An important predefined *Rect* is *GeoNullRect*, the rectangle with both its lower-left and upper-right at the origin (0, 0). If linked lists of *Rect*s are needed, the *LinkedRect* primitive can be used.

Another primitive is a position relative to a point (*GEO_NORTH*, *GEO_EAST*, etc). There are a total of nine positions, corresponding to the eight points around a single point in a grid plus the point itself (*GEO_CENTER*).

The final primitive is a *Transform*, which represents some combination of rotation by a multiple of 90 degrees, mirroring across the *x* or *y* axis, scaling by an integer scale factor, and translation by an integer *x* and *y* displacement. A *Transform* can be thought of as representing a simple linear transformation on two-dimensional points, or as a matrix of the form:

$$\begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

Multiplying a point vector of the form *(x, y, 0)* by this transform gives a transformed point *(x', y', 0)*. Although the transform matrix has nine elements, the three on the right-hand are always constant, so only six numbers are needed to describe a transform: four for the rotation (*a*, *b*, *d*, *e*) and two for the translation (*c*, *f*). Because the only rotations are multiples of 90 degrees, transforms will always be of one of the following even more specific forms (only the four rotation numbers are shown), where *S* is the integer scale factor:

$$\begin{matrix} S & 0 \\ 0 & S \end{matrix} \qquad \begin{matrix} 0 & -S \\ S & 0 \end{matrix} \qquad \begin{matrix} -S & 0 \\ 0 & -S \end{matrix} \qquad \begin{matrix} 0 & S \\ -S & 0 \end{matrix}$$

$$\begin{matrix} S & 0 \\ 0 & -S \end{matrix} \qquad \begin{matrix} 0 & S \\ S & 0 \end{matrix} \qquad \begin{matrix} -S & 0 \\ 0 & S \end{matrix} \qquad \begin{matrix} 0 & -S \\ -S & 0 \end{matrix}$$

The first four forms correspond to clockwise rotations of 0, 90, 180, and 270 degrees, and the second four correspond to the same four orientations flipped upside down (mirror across the *x*-axis after rotating).

The above rotations or mirrorings with a scale factor of 1 exist as predefined transforms. *GeoIdentityTransform* is the identity transformation, i.e, no transformation at all, or the first transform listed above. *Geo90Transform*, *Geo180Transform*, and *Geo270Transform* correspond to the next three transformations, or clockwise rotations of 90, 180, and 270 degrees respectively. *GeoUpsideDownTransform* is the next transform, mirroring across the *x*-axis. *GeoSidewaysTransform* is the seventh transform, corresponding to mirroring across the *y*-axis. The remaining two transforms above (the sixth and eighth) don't have any predefined transforms, but can be built by composing predefined transforms using *GeoTransTrans* (see below).

A number of macros exist for determining relationships between *Point*s and *Rect*s. *GEO_OVERLAP* is TRUE if two rectangles share some area in common. *GEO_TOUCH* is TRUE if two rectangles share some area or any part of their perimeters (including touching only at a corner). *GEO_SURROUND* is TRUE if *r1* completely surrounds *r2*, where the boundaries of *r1* and *r2* are allowed to touch. *GEO_SURROUND_STRONG* is like *GEO_SURROUND*, but is only TRUE if *r1* completely surrounds *r2* without their borders touching. *GEO_ENCLOSE* is TRUE if a point *p* lies inside or on the border of the rectangle *r*. *GEO_RECTNULL* is TRUE if *r* has zero area, which can result if the *x*-coordinate of its upper-right is less than or equal to the *x*-coordinate of its lower-left, or similarly for the *y*-coordinates. Finally, *GEO_EXPAND* is used to grow (or shrink) a rectangle *src* by an integer distance *amount*, leaving the new rectangle in *dst* (which may be the same as *src*).

Many procedures exist to manipulate transformations. In general, when they accept more than one Point or Rect as arguments, the Points or Rects must be distinct from each other (i.e, no aliasing is allowed). *GeoTransPoint* applies the Transform **t* to the Point **psrc* and leaves its result in the Point **pdst*. *GeoTransRect* is identical, but for Rects; it applies *t* to **rsrc* and leaves its result in **rdst*. *GeoTransRect* guarantees that *rdst->r_ur* is really above and to the right of *rdst->r_ll*, by interchanging upper and lower coordinates if necessary after the transform. Note that this is NOT the same as transforming the upper-right and lower-left Points separately, since separate transformations can result in a rectangle whose upper right is below its lower left (e.g, *GeoUpsideDownTransform*).

Three procedures compose transforms, producing the transform that is equivalent to applying first one, then the second of the two transforms. There are two special-case procedures. *GeoTranslateTrans* composes first the Transform **tsrc* and then a simple translation by *x* and *y*, storing its result in **tdst*. *GeoTransTranslate* composes first a simple translation by *x* and *y*, followed by the Transform **tsrc*, also storing its result in **tdst*. Finally, *GeoTransTrans* composes two arbitrary transforms **t1* and **t2*, leaving its result in **tdst*.

Transforms that adhere to one of the eight rotation formats described above are always invertible. The inverse of such a transform can be computed by *GeoInvertTrans*, which leaves the inverse of **tsrc* in **tinv*.

Two procedures extract useful information from Transforms. *GeoScale* returns the scale factor associated with the Transform **t*. *GeoDecomposeTransform* breaks up a transform into an optional mirror about the x-axis (i.e., flipping upside down), followed by an optional counterclockwise rotation. It sets **upsidedown* to **TRUE** if the transform requires flipping upside down before rotation, and sets **angle* to the degrees of rotation: 0, 90, 180, or 270.

Three procedures manipulate positions such as *GEO_NORTH*. *GeoNameToPos* maps the ASCII *name* for a position (e.g, ''north'', ''top'', or ''left'', ''west'', etc) into the internal position number. If *name* is ambiguous, -1 is returned; if *name* is unrecognized, -2 is returned. If *manhattan* is TRUE, only the directions corresponding to *GEO_NORTH*, *GEO_SOUTH*, *GEO_WEST*, or *GEO_EAST* are accepted. If *printerrors* is TRUE, *GeoNameToPos* will print an error message on the standard output in addition to returning -1 or -2. The inverse of *GeoNameToPos* is *GeoPosToName*, which returns the ASCII string for a given position *pos*. *GeoTransPos* applies the Transfor **t* to the position *pos* and returns the new position. Only the rotational part of **t* is relevant; the translation is ignored.

The next collection of procedures manipulate Points and Rects. *GeoInclude* and *GeoIncludeAll* extend whichever sides of the Rect **dst* that are necessary to include the area of the Rect **src*. Both return TRUE if **dst* was enlarged. If **src* is considered to be zero-size (see below), **dst* is unchanged. If **dst* is zero-size, it is set to **src* if **src* is not also zero-size. The two procedures differ in that *GeoInclude* considers zero-area rectangles to be zero-size, while *GeoIncludeAll* only considers rectangles whose bottom is actually above their top or whose LHS is to the right of their RHS to be zero-size. *GeoIncludePoint* is like *GeoInclude* except **src* is a Point instead of a Rect.

Three procedures are provided for clipping. *GeoClip* determines the portion of the Rect **r* that overlaps the Rect **cliparea* and replaces **r* with the new Rect. If **r* and **cliparea* don't overlap at all, **r* is turned inside out (*r_xbot* > *r_xtop* or *r_ybot* > *r_ytop*). *GeoClipPoint* moves the Point **p* to the closest point on the boundary of the Rect **cliparea* if it isn't already contained in **cliparea* or on its border. Finally, *GeoDisjoint* is used to clip a Rect against another, but to apply a procedure to each region in **area* that lies outside **cliparea*, instead of modifying **area*. The procedure *(*proc)()* it applies should be like the library procedure *GeoDummyFunc*, which accepts a Rect and the *cdata* argument passed to *GeoDisjoint* and returns TRUE always. If *(*proc)()* returns FALSE, *GeoDisjoint* aborts and returns FALSE itself; otherwise, it returns TRUE. *GeoDisjoint* works in ''tile'' space, so each rectangle is considered to contain its lower *x*- and *y*-coordinates, but not its upper coordinates.

The discussion earlier on transformation mentioned that transforming the two corner points of a Rect independently could result in a Rect whose lower left was above or to the right of its upper right. *GeoCanonicalRect* can remedy this situation; it flips the top and bottom or left and right (or both) of the Rect **rsrc* as necessary to ensure that the upper right is above and to the right of the lower left, leaving the canonical Rect in **rdst.*

Two procedures compute the relative positions of Points and Rects. *GeoRectPointSide* gives the side (*GEO_NORTH*, etc) of the Rect **r* on which the Point **p* lies (**p* must lie on the boundary of **r*; otherwise, *GEO_CENTER* is returned). Similarly, *GeoRectRectSide* gives the side of **r1* on which **r2* lies, or *GEO_CENTER* if they don't share any side. Unfortunately this procedure doesn't detect the case where the Rects share a coordinate without sharing a side (e.g, the LHS of one is equal to the RHS of the other, but they don't come even close in the vertical dimension).

A final procedure is provided for high-speed reading of ascii files containing descriptions of rectangles, *GetRect*. This procedure reads from a stdio-opened FILE **f*, which should be positioned so that after skipping *nskip* characters, it will be at the start of a line containing four ascii numbers that will be stored in *r->r_xbot*, *r->r_ybot*, *r->r_xtop*, and *r->r_ytop*. It returns TRUE if it successfully recognized a rectangle, FALSE on error or end-of-file. *GetRect* is considerably faster than either *fscanf*(3s) or even *fgets*(3s) followed by manual decoding of the line, because it reads data directly from the stdio buffer in its input file. As such, it depends on the structure of a FILE, and may fail to work properly on machines with wildly different implementations of the stdio library from the standard Berkeley distribution (those in which certain fields are nonexistent or renamed).

MACROS FOR SPEED

If speed is essential, macros are defined in **geofast.h** to take the place of the several procedures for special cases. *GEOCLIP* is identical to the procedure *GeoClip*, but it returns no value. Four macros for manipulating Transforms, *GEOTRANSRECT*, *GEOTRANSTRANS*, *GEOINVERTTRANS*, and *GEOTRANSTRANSLATE*, are similar to their procedural counterparts *GeoTransRect*, *GeoTransTrans*, *GeoInvertTrans*, and *GeoTransTranslate*, but only work with Transforms whose scale factor is unity (1). These macros are several times faster than their procedural counterparts; on a Sun-2 the speed difference is close to a factor of 10, but on other machines the difference is less extreme.

SEE ALSO

magicutils(3)

NAME

hash – procedures for managing hash tables in libmagicutils.a

SYNOPSIS

```
#include hash.h

HashInit(table, initsize, keysize)
HashTable *table;
int initsize, keysize;

HashInitClient(table, initsize, keysize, compareFn, copyFn, hashFn, killFn)
HashTable *table;
int initsize, keysize;
int (*compareFn)(key1, key2);
char *(*copyFn)(key);
int (*hashFn)(key);
Void (*killFn)(key);

int HashSize(keybytes)
int keybytes;

HashKill(table)
HashTable *table;

HashEntry *HashLookOnly(table, key)
HashTable *table;
ClientData key;

HashEntry *HashFind(table, key)
HashTable *table;
ClientData key;

ClientData HashGetValue(he)
HashEntry *he;

HashSetValue(he, value)
HashEntry *he;
ClientData value;

HashStartSearch(hs)
HashSearch *hs;

HashEntry *HashNext(table, hs)
HashTable *table;
HashSearch *hs;
```

DESCRIPTION

This module provides procedures for creating, accessing, and destroying hash tables. These tables grow automatically as more elements are added to them to avoid overloading. They may be indexed by strings, single words, or multi-word structures. Single-word can be interpreted (e.g., compared or hashed) by user-supplied procedures. Each entry stores a single word value, which may be set or read by the macros *HashSetValue* or *HashGetValue* but should not be manipulated directly.

HashInit is used to allocate space for the initially empty hash table *table*. Enough space is allocated for *initsize* buckets (which should be a power of two), although subsequent additions to the hash table can cause the number of buckets to increase. Tables can be organized in one of three different ways, depending on the value of *keysize*.

If *keysize* is **HT_STRINGKEYS**, then keys passed to *HashFind* (or *HashLookOnly*) are treated as the addresses of NULL-terminated strings. The *HashEntry* structures for this type of key are variable-sized; sufficient space is allocated at the end of each structure to hold the key string and its trailing NULL byte. If *keysize* is **HT_WORDKEYS**, then keys are single words of data passed directly to *HashFind*, and are compared using *strcmp*(1). If *keysize* is **HT_STRUCTKEYS** or greater, keys are multiple words of data, but their address is passed to *HashFind* (instead of the actual value as when *keysize* was **HT_WORDKEYS**). The value of *keysize* in this case should be the number of words in a key. The macro *HashSize* should be used to produce this number; *HashSize*(**sizeof** (**struct** *foo*)) gives the number of words needed if keys are to be of type (*struct foo*). In general, single-word keys (*keysize* equal to **HT_WORDKEYS**) are the fastest, but the most restrictive.

A second procedure, *HashInitClient*, may be used to initialize a hash table instead of *HashInit*. This second procedure is a more general one, in that it allows a fourth value of *keysize* to be provided, **HT_CLIENTKEYS**, along with four client procedures. The keys in such a case are single-word values, passed to *HashFind* just like keys when *keysize* is **HT_WORDKEYS**. However, they are interpreted using the client procedures passed in the call to *HashInitClient*. These procedures perform four functions; if any are NULL, then those functions are performed exactly as in the case of **HT_WORDKEYS**. The first, *(*compareFn)(key1, key2)*, takes two single-word key values and returns either 0 if they are equal, or 1 if they are not. The next procedure, *(*copyFn)(key)*, is called when a new entry is being created for a key; it performs whatever processing is needed to ensure that the key can be kept around permanently (e.g., making a copy of it), and returns the value that will actually be stored as the key in the hash table (e.g., the copy). The third procedure, *(*hashFn)(key)*, is used to produce a single 32-bit value from *key*. It is primarily useful when *key* is in fact a pointer to a structure, and the contents of the structure, rather than its address, determine the hash value. Finally, *(*killFn)(key)* is called when the hash table is being freed by *HashKill* to perform any final cleanup of a key, such as freeing a key that had been copied by *(*copyFn)()* when it was installed in the hash table.

HashKill can be used to free all the storage associated with *table*.

Both *HashLookOnly* and *HashFind* are used for retrieving the entry from *table* that matchs *key*. They differ in their behavior when *key* is not in the table. *HashLookOnly* will return NULL if the key is not found, while *HashFind* will create a new HashEntry whose value (as returned by *HashGetValue*) is zero.

It is possible to scan sequentially through a hash table to visit all its entries. *HashStartSearch* initializes the *HashSearch* structure *hs*, which is then passed to *HashNext*, which keeps returning subsequent entries in the table until all have been returned, when it returns NULL.

BUGS

If it is possible for initialized entries in the hash table to have NULL values, then *HashLookOnly* must be called before *HashFind* if you are to be certain that an entry was not already in the table, since there is no distinction between a NULL value that was already in the table and a NULL value that signifies that the entry was newly created by *HashFind*.

SEE ALSO

magicutils(3)

NAME

heap – procedures for managing sorted heaps in libmagicutils.a

SYNOPSIS

```
#include magic.h
#include heap.h

typedef struct { int he_key, char *he_id; } HeapEntry;

bool HEAP_EMPTY(h)
Heap *h;

HeapInit(h, initsize, descending, stringids)
Heap *h;
int initsize;
bool descending, stringids;

HeapKill(h, func)
Heap *h;
int (*func)(h, index);

HeapFreeIdFunc(h, index)
Heap *h;
int index;

HeapEntry *HeapRemoveTop(h, entry);
Heap *h;
HeapEntry *entry;

HeapAdd(h, key, id)
Heap *h;
int key;
char *id;
```

DESCRIPTION

These procedures create, manipulate, and destroy heaps. A heap is essentially an array that automatically sorts itself when items are added to it. The items added to the heap consist of an integer key and a one-word datum which can either be the address of a NULL-terminated string (treated specially), or any other one-word data item. Heaps can be sorted in either ascending or descending order. The data storage for a heap automatically grows as more elements are added to the heap.

The *HeapEntry* structure identifies the integer key value (*he_key*) on which the element is sorted, and a one-word datum (*he_id*). Heaps are created by *HeapInit*, which initializes the data storage for *h*. Enough space is left initially for *initsize* elements, although the heap will grow automatically as more elements are added. If *descending* is **TRUE**, the largest element in the heap will be removed first; otherwise, the smallest element will be the first to be removed by *HeapRemoveTop*. Each heap entry has an associated datum or *id*; if *stringids* is TRUE, these are considered to be ASCII strings and handled specially by *HeapAdd* and *HeapKill*.

HeapKill deallocates the storage associated with a heap. If *func* is non-NULL, it is applied to each element in the heap. A common use of *func* is to free the storage associated with string ids in the heap, such as is necessary when the heap was created with *stringids* set to **TRUE** in the call to *HeapInit* above. A library function, *HeapFreeIdFunc*, is provided for this purpose.

HeapRemoveTop places the top element from *h* in the *HeapEntry* pointed to by *entry* and returns *entry*. However, if the heap was empty, *HeapRemoveTop* returns NULL. *HeapRemoveTop* always removes the smallest (if keys are ascending) or largest (if keys are descending) element from the heap.

HeapAdd is used to add a new entry to *h*. The new entry has an integer key of *key*, and a value of *id*. If the heap was created with *stringids* to be **TRUE** in *HeapInit*, then *id* is interpreted as a NULL-terminated ASCII string; sufficient additional memory to hold this string is allocated, the string is copied into this new memory, and a pointer to this new memory is stored with the heap entry. Otherwise, the value of *id* is just stored directly in the heap entry.

BUGS

The management of the *he_id* field should be consistent with the management of keys for hash tables, i.e, multi-word structures should be supported along with strings and single-word values.

SEE ALSO

magicutils (3)

NAME

list – procedures for managing lisp style lists in libmagicutils.a

SYNOPSIS

#include magic.h
#include list.h

LIST_ADD(item,list)

LIST_FIRST(list)

LIST_TAIL(list)

bool ListContainsP(element, list)
ClientData element;
List *list;

Void ListDealloc(list)
List *list;

Void ListDeallocC(list)
List *list;

int ListLength(list)
List *list;

ClientData ListPop(listPP)
List **listPP;

List *ListReverse(list)
List *list

DESCRIPTION

These macros and procedures permit the implementation of linked lists of arbitrary things. The lists are lisp like, i.e., list pointers are in separate structures rather than in the strucs being linked. Macros are distinguished from procedues by names that are all upper-case.

LIST_ADD(i,l) adds an item to the front of a list.

LIST_COPY(l,lnew) creates a copy of a list

LIST_FIRST references the first item on the list.

LIST_TAIL(l) references the sublist consisting of all but the first item of the list.

ListContainsP returns **TRUE** in the specified item is contained in the list.

int ListLength returns the length of the list.

Void ListDealloc reclaims a list (but not its contents).

Void ListDeallocC reclaims a list /fIand/fR its contents.

ListPop deletes the first item from the list, and returns it (the item).

*List *ListReverse* creates and returns a reversed copy of a list.

SEE ALSO

magicutils (3)

NAME

magicutils – collection of utility procedures in -lmagicutils

SYNOPSIS

cc -I~*cad*/**src/magic/include** ~*cad*/**src/magic/lib/libmagicutils.a**
cc -I~*cad*/**src/magic/include** ~*cad*/**src/magic/lib/libmagictrace.a**
cc -I~*cad*/**src/magic/include -pg** ~*cad*/**src/magic/lib/libmagicutils_p.a**
cc -I~*cad*/**src/magic/include -pg** ~*cad*/**src/magic/lib/libmagictrace_p.a**
(replace ~*cad* with the home directory of the user **cad**).

MainExit(code) int code;

TxError(fmt, va_alist) char *fmt; va_dcl;

char *TxGetLine(buf, len) char *buf; int len;

DESCRIPTION

The two libraries *libmagicutils.a* and *libmagictrace.a* include all of the procedures from the *utils* module used internally by the Magic layout system. The first library is for normal use; the second library is for use with the tracing option of the new memory allocator. See the documentation on the individual pieces of the library for details of the procedures they contain.

To use these libraries, you should compile your programs with the flag **-I**~*cad*/**src/magic/include** (to search the Magic include directory for needed **.h** files). The documentation for the various pieces of the libraries lists which **.h** files are needed for which procedures.

Three default procedures are defined for the library but can be replaced by your own procedures if you so wish. The procedures are *MainExit*, which has the same semantics as *exit* (3) but can be replaced by your own procedure by that name to do additional cleanup, *TxError*, which is like *fprintf(stderr, fmt, args)*, where *args* can be zero or more arguments, just as in *fprintf* (3), and finally *TxGetLine*, which is like *fgets(buf, len, stdin*). The library versions of these procedures only get pulled in if you haven't defined them yourself.

Versions exist of both libraries with profiling (**-pg**) enabled; these are *libmagicutils_p.a* and *libmagictrace_p.a*.

SEE ALSO

magic(1), dqueue(3), geometry(3), hash(3), heap(3), list(3), malloc(3), path(3), runstats(3), set(3) show(3) stack(3), string(3)

NAME

mallocMagic, freeMagic – a new memory allocator in libmagicutils.a

SYNOPSIS

#include magic.h
#include malloc.h

char *mallocMagic(size)
unsigned size;

char *callocMagic(size)
unsigned size;

MALLOC(*type_decl***, var, size)**
type_decl **var;**
unsigned size;

CALLOC(*type_decl***, var, size)**
type_decl **var;**
unsigned size;

freeMagic(var)
char *var;

FREE(var)
char *var;

cc -DMALLOCTRACE ... ~cad/src/magic/lib/libmagictrace.a

mallocTraceInit(filename)
char *filename;

mallocTraceEnable()

mallocTraceDisable()

mallocTraceDone()

mallocTraceOnlyWatched(only)
bool only;

bool mallocTraceWatch(addr)
char *addr;

bool mallocTraceUnWatch(addr)
char *addr;

DESCRIPTION

These procedures implement a new memory allocator. They provide fast allocation and freeing for programs that allocate thousands or millions of objects of similar sizes. Speed results from maintaining separate free-lists for objects of each size, providing fast macros *MALLOC* and *FREE* for doing allocation and freeing, and clustering objects of the same size on the same page in an attempt to improve locality of reference. In addition, these procedures provide features to aid in the debugging of programs that do a lot of memory allocation and freeing; used in conjunction with *prleak* (8) they can detect storage leaks and also duplicate attempts to free the same storage location.

Memory is allocated using either the procedure *mallocMagic* or the macro *MALLOC*. The former has an interface identical to that of the standard UNIX library procedure *malloc* (3), namely, it returns a pointer to a region of memory sufficiently large to hold *size* bytes. The macro *MALLOC* is noticeably faster, particularly on machines with brain-dead procedure calls (such as a certain popular machine made by the second largest U.S. computer manufacturer). Its usage is a bit unusual, in that its first argument is a type

specification and its second is modified in place. For example, to allocate an object of type *HashEntry* * that is 20 bytes long, and to assign this to the pointer *he*, one could write:

```
MALLOC(HashEntry *, he, 20);
```

Note that there are no parentheses around the **HashEntry *** above. After executing this macro, *he* would point to a *HashEntry* that was 20 bytes long.

The macro *CALLOC* and the procedure *callocMagic* perform function analagous to *MALLOC* and *mallocMagic* except that the malloc'd memory is zeroed.

Memory can be freed using either the procedure *freeMagic*, which frees its argument *var* exactly as does the UNIX *free* (3), or using the *FREE* macro, which does the same thing to *var* but is faster.

Users of *MALLOC* and *FREE* should beware that they are macros that include C statements enclosed in a pair of braces (**{ ... }**), and should be treated accordingly. For example, it is not legal to type:

```
if (i != j)
        MALLOC(HashEntry *, he, i);
else
        return (NULL);
```

One should instead use:

```
if (i != j)
{
        MALLOC(HashEntry *, he, i);
}
else
        return (NULL);
```

If you wish to take advantage of the debugging features of this memory allocator, you must do two things. First, compile all of your **.c** files that #include ''malloc.h'' with the **-DMALLOCTRACE** flag. Second, when you link your program to build an *a.out* file, use the library **~cad/src/magic/lib/libmagictrace.a** instead of the normal **libmagicutils.a**. The **libmagictrace.a** library contains additional code to maintain the information needed by the debugging procedures below. If you link your program with the standard library, it will link successfully, but the debugging procedures won't do anything.

The debugging procedures produce a trace file for subsequent analysis by *prleak* (8). Before any memory is allocated, you should call *mallocTraceInit* to create the trace file *name*. Tracing won't actually begin, however, until you call *mallocTraceEnable*. From that point until *mallocTraceDisable*, all calls to *mallocMagic* or *freeMagic* (or their corresponding macro versions *MALLOC* and *FREE*) will be logged to the trace file. Calls to *mallocTraceDisable* and *mallocTraceEnable* may be nested; only the outermost *mallocTraceEnable* has any effect.

If more selective tracing is desired, you can specify that trace information is to be output only for certain addresses. Calling *mallocTraceOnlyWatched* with *only* equal to **TRUE** causes this to happen. An address *addr* is added to the list of addresses to trace by calling *mallocTraceWatch*, or removed from this list by calling *mallocTraceUnWatch*. When *mallocTraceOnlyWatched* is called with *only* equal to **FALSE**, operation reverts to the normal mode of tracing all addresses.

When you are finished with all memory allocation tracing and want to flush all results out to the trace file, call *mallocTraceDone*. Subsequent calls to the memory allocator will not be traced.

BUGS

The *MALLOC* and *FREE* macros are syntactically clumsy, but unfortunately some C optimizers have trouble with syntactically cleaner forms.

The ability to trace specific addresses is only useful if you know which ones to watch. A more generally useful facility would probably be to watch certain sizes of objects, or to allow the user to supply a procedure that could determine whether or not an address was to be traced.

SEE ALSO

magicutils (3)

NAME

mpack – routines for generating semi-regular modules

DESCRIPTION

Mpack is a library of 'C' routines that aid the process of generating semi-regular modules. Decoder planes, barrel shifters, and PLAs are common examples of semi-regular modules.

Using Magic, an mpack user will draw an example of a finished module and then break it into tiles. These tiles represent the building blocks for more complicated instances of the module. The mpack library provides routines to aid in assembling tiles into a finished module.

MAKING AN EXAMPLE MODULE

The first step in using mpack is to create an example instance of the module, called a *template*. The basic building blocks of the structure, or *tiles*, are then chosen. Each tile should be given a name by means of a rectangular label which defines its contents. If the tiles in the module do not abut (e.g. they overlap) it is useful to define another tile whose size indicates how far apart the tiles should be placed.

Templates should be in Magic format and, by convention, end with a **.mag** suffix. With some programs, it is possible to generate the same structure in a different technology or style by changing just the template. If this is the case, each template should have a filename of the
form basename-*style***.mag**. The *style* part of the filename interacts with the **-s** option (see later part of this manual).

WRITING AN MPACK PROGRAM

An mpack program is the 'C' code which assembles tiles into the desired module. Typically this program reads a file (such as a truth table) and then calls the tile placement routines in the mpack library.

The mpack program must first include the file **~cad/lib/mpack.h** which defines the interface to the mpack system. Next the **TPinitialize** procedure is called. This procedure processes command line arguments, opens an input file as the standard input (**stdin**), and loads in a template.

The program should now read from the standard input and compute where to place the next tile. Tiles may be aligned with previously placed tiles or placed at absolute coordinates. If a tile is to overlap an existing tile the program must space over the distance of the overlap before placing the tile.

When all tiles are placed the program should call the routine **TPwrite_tile** to create the output file that was specified on the command line.

To use the mpack library be sure to include it with your compile or load command (e.g. **cc** *your_file* **~cad/lib/mpack.lib**).

ROUTINES

Initialization and Output Routines

TPinitialize(*argc, argv, base_name***)**

The mpack system is initialized, command line arguments are processed, and a template is loaded. The file descriptor **stdin** is attached to the input file specified on the command line. The template's filename is formed by taking the *base_name*, adding any extension indicated by the **-s** option, and then adding the **.mag** suffix. The **-t** option allows the user to override *base_name* from the command line.

Argc and *argv* should contain the command line arguments. *Argc* is a count of the number of arguments, while *argv* is an array of pointers to strings. Strings of length zero are ignored (as is the flag consisting of a single space), in order to make it easy for the calling program to intercept its own arguments. *Argc* and *argv* are of the same structure as the two parameters passed to the main program. A later section of this manual summarizes the command line options.

TPload_tiles(*file_name***)**

The given *file_name* is read, and each rectangular label found in the file becomes a tile accessible via TPname_to_tile. No extensions are added to *file_name*.

TILE TPread_tile(*file_name***)**
A tile is created and *file_name* is read into it. The tile is returned as the value of the function.

TPwrite_tile(*tile, filename***)**
The tile *tile* is written to the file specified by *filename*, with **.ca** or **.cif** extensions added. See the description of the **-o** option for information on what file name is chosen if *filename* is the null string. The choice between Magic or CIF format is chosen with the **-a** or **-c** command line options.

Tile creation, deletion, and access

TPdelete_tile(*tile***)**
The tile *tile* is deleted from the database and the space occupied by it is reused.

TILE TPcreate_tile(*name***)**
A new, empty tile is created and given the name *name*. This name is used by the routine **TPname_to_tile** and in error messages. The type **TILE** returned is a unique ID for the tile, not the tile itself. Currently this is implemented by defining the type TILE to be a pointer to the internal database representation of the tile.

int TPtile_exists(*name***)**
TRUE (1) is returned if a tile with the given *name* exists (such as in the template or from a call to TPcreate_tile).

TILE TPname_to_tile(*name***)**
A value of type **TILE** is returned. This value is a unique ID for the tile that has the name *name*. This name comes from a call to TPcreate_tile(), or from the rectangular label that defined it in a template that was read in by TPread_tiles() or TPinitialize(). If the tile does not exist then a value of NULL is returned and an error message is printed.

RECTANGLE TPsize_of_tile(*tile***)**
A rectangle is returned that is the same size as the tile *tile*. The rectangle's lower left corner is located at the coordinate (0, 0). All coordinates in mpack are specified in half-lambda.

Painting and Placement Routines

RECTANGLE TPpaint_tile(*from_tile, to_tile, ll_corner***)**
The tile *from_tile* is painted into the tile *to_tile* such that its lower left corner is placed at the point *ll_corner* in the tile *to_tile* . The location of the newly painted area in the output tile is returned as a value of type RECTANGLE. The tile *to_tile* is often an empty tile made by **TPcreate_tile()**. The point *ll_corner* is almost never provided directly, it is usually generated by routines such as **align()**.

TPdisp_tile(*from_tile, ll_corner***)**
A rectangle the size of *from_tile* with the lower left corner located at *ll_corner* is returned. Note that this routine behaves exactly like the routine TPpaint_tile except that no output tile is modified. This routine, in conjunction with the **align** routine, is useful

for controlling the overlap of tiles.

RECTANGLE TPpaint_cell(*from_tile, to_tile, ll_corner***)**
This routine behaves like **TPpaint_tile()** except that the *from_tile* is placed as a subcell rather than painted into place. The tile *from_tile* must exist in the file system (i.e. it must have been read in from disk or have been written out to disk).

Label Manipulation Routines

TPplace_label(*tile, rect, label_name***)**
A label named *label_name* is place in the tile *tile*. The size and location of the label is the given by the RECTANGLE *rect*.

int TPfind_label(*tile, &rect1, str, &rect2***)**
The tile *tile* is searched for a label of name *str*. The location of the first such label found is returned in the rectangle *rect2*. The function returns 1 if such a label was found, and 0 otherwise. The rectangle pointer *&rect1*, if non-NULL, restricts the search to an area of the tile.

TPstrip_labels(*tile, ch***)**
All labels in the tile *tile* that begin with the character *ch* are deleted.

TPremove_labels(*tile, name, r***)**
All labels in the tile *tile* that are completely within the area *r* are deleted. If *name* is non-NULL, then only labels with that name will be affected.

TPstretch_tile(*tile, str, num***)**
The string *str* is the name of one or more labels within the tile *tile*. Each of these labels must be of zero width or zero height, i.e. they must be lines. Each of these lines define a line across which the tile will be stretched. The amount of the stretch is specified by *num* in units of **half**-lambda. Stretching such a line turns it into a rectangle. Note that if the tile contains 2 lines that are co-linear, the stretching of one of them will turn both into rectangles.

Point-Valued Routines

POINT tLL(*tile***)**
POINT tLR(*tile***)**
POINT tUL(*tile***)**
POINT tUR(*tile***)**
The location of the specified corner of tile *tile*, relative to the tile's lower left corner, is returned as a point. LL stands for lower-left, LR for lower-right, UL for upper-left, and UR for upper-right. Note that **tLL()** returns (0, 0).

POINT rLL(*rect***)**
POINT rLR(*rect***)**
POINT rUL(*rect***)**
POINT rUR(*rect***)**
The location of the specified corner of the rectangle *rect* is returned as a point. LL stands for lower-left, LR for lower-right, UL for upper-left, and UR for upper-right.

POINT align(*p1, p2***)**

A point is computed such that when added to the point *p2* gives the point *p1*. *p1* is normally a corner of a rectangle within a tile and *p2* is normally a corner of a tile. In this case the point computed can be treated as the location for the placement of the tile.

For example, TPpaint_tile(outtile, fromtile, align(rUL(rect), tLL(fromtile))) will paint the tile *fromtile* into *outtile* such that the lower left corner of *fromtile* is aligned with the upper-left corner of *rect*. In this example *rect* would probably be something returned from a previous TPpaint_tile() call.

Point and Rectangle Addition Routines

POINT TPadd_pp(*p1, p2***)**
POINT TPsub_pp(*p1, p2***)**
The points *p1* and *p2* are added or subtracted, and the result is returned as a point. In the subtract case *p2* is subtracted from *p1*.

RECTANGLE TPadd_rp(*r1, p1***)**
RECTANGLE TPsub_rp(*r1, p1***)**
The rectangle *r1* has the point *p1* added or subtracted from it. This has the effect of displacing the rectangle in the X and/or Y dimensions.

Miscellaneous Functions

int TPget_lambda()
This function returns the current value of lambda in centi-microns.

INTERFACE DATA STRUCTURES

In those cases where tiles must be placed using absolute, (half-lambda) coordinates, it is useful to know that **RECTANGLE**s and **POINT**s are defined as:

```
typedef struct {
   int x_left, x_right, y_top, y_bot;
} RECTANGLE;

typedef struct {
   int x, y;
} POINT;
```

The variable **origin_point** is predefined to be (0, 0). **origin_rect** is defined to be a zero-sized rectangle located at the origin.

OPTIONS ACCEPTED BY TPinitialize()

Typical command line: *program_name* [-t *template*] [-s *style*] [-o *output_file*] *input_file*

-a produce Magic format (this is the default)

-c produce CIF format

-v be verbose (sequentially label the tiles in the output for debugging purposes; also print out information about the number of rectangles processed by mpack)

-s *style* generate output using the template for this style (see TPinitialize for details)

-o The next argument is taken to be the base name of the output file. The default is the input file name with any extensions removed. If there is not input file specified and no -o option specified, the output will go to **stdout**.

-p (pipe mode) Send the output to **stdout**.

-t The next argument specifies the template base name to use. This overrides the default supplied by the program. A **.mag** extension is automatically added. (see TPinitialize)

-l *name* Set the cif output style to *name*. *name* is the name of a cif output style as defined in Magic's technology file. If this option is not specified then the first output style in the technology file is used. (Note: In the old tpack system this option set the size of lambda.)

input_file
The name of the file that the program should read from (such as a truth table file). If this filename is omitted then the input is taken from the standard input (such as a pipe).

-M *num* This option is accepted by mpack, but ignored. It is a leftover from the tpack system.

-D *num1 num2*
The *Demo* or *Debug* option. This option will cause **mpack** to place only the first *num1* tiles, and the last *num2* of those will be outlined with rectangular labels. In addition, if a tile called "blotch" is defined then a copy of it will be placed in the output tile upon each call to the *align* function during the placing of the last *num2* tiles. The blotch tile will be centered on the first point passed to *align*, and usually consists of a small blotch of brightly colored paint. This has the effect of marking the alignment points of tiles. The last tile painted into is assumed to be the output tile.

EXAMPLE

It is highly recommended that the example in **~cad/src/mquilt** be examined. Look at both the template and the 'C' code. A more complex example is in **~cad/src/mpla**.

FILES

~cad/lib/mpack.h (definition of the mpack interface)
~cad/lib/mpack.lib (linkable mpack library)
~cad/lib/mpack.ln (lint-library for lint)
~cad/src/mquilt/* (an example of an mpack program)
~cad/lib/magic/sys/*.tech* (technology description files)

ALSO SEE

magic(CAD), mquilt(CAD), mpla(CAD)
Robert N. Mayo *Pictures with Parentheses: Combining Graphics and Procedures in a VLSI Layout Tool*, ®Proceedings of the 20th Design Automation Conference, June, 1983.
`C' Manual

HISTORY

This is a port of the tpack(1) system which generated Caesar files.

AUTHOR

Robert N. Mayo

BUGS

When a tile contains part of a subcell, or touches a subcell, then the whole subcell is considered to be part of the tile. The same goes for arrays of subcells.

NAME

path – procedures for managing search paths in libmagicutils.a

SYNOPSIS

```
#include <stdio.h>
#include utils.h

int PaConvertTilde(psource, pdest, size)
char **psource, **pdest;
int size;

FILE *PaOpen(file, mode, ext, path, libpath, prealname)
char *file, *mode, *ext;
char *path, *libpath;
char **prealname;

char *PaSubsWD(path, newWD)
char *path, *newWD;

int PaEnum(path, file, func, cdata)
char *path, *file;
int (*func)(name, cdata);
ClientData cdata;
```

DESCRIPTION

These procedures implement a path mechanism, whereby several places may be searched for files.

PaConvertTilde is used to convert *csh* (1)-style tilde notation for users' home directories (e.g., ''~wss'', ''~/mydir/file.o'') to standard directory names as understood by *open* (2), etc. If ***psource* is a tilde (''~''), then the name following the tilde up to the first slash or end of string is converted to a home directory and stored in the string pointed to by **pdest*. Then remaining characters in the file name at **psource* are copied to **pdest* (the file name is terminated by white space, a NULL character, or a colon) and **psource* is updated. Upon return, **psource* points to the terminating character in the source file name, and **pdest* points to the null character terminating the expanded name. If a tilde cannot be converted because the user name cannot be found, **psource* is still advanced past the current entry, but nothing is stored at the destination. At most *size* characters (including the terminating null character) will be stored at **pdest*. The name consisting of a single tilde, i.e, ''~'' with no user name, expands to the current user's home directory. *PaConvertTilde* returns the number of bytes of space left in the destination area if successful, or -1 if the user name couldn't be found in the password file.

PaOpen opens a file, looking it up in the current path and supplying a default extension. It either returns a pointer to a *FILE*, as does *fopen* (3s), or NULL if no file could be opened. The mode of the file opened is determined by *mode*, also as in *fopen*. If *ext* is specified, then it is tacked onto the end of *name* to construct the name of the file *PaOpen* will attempt to find. (*Ext* must begin with a dot if that is the extension separator; none is inserted automatically.) If the first character of *name* is a tilde or slash, *PaOpen* tries to look up the file with the original name (and extension), doing tilde expansion if necessary and returning the result. Otherwise, it goes through the search path *path* (a colon-separated list of directories much as in *csh* (1)) one entry at a time, trying to look up the file once for each path entry by prepending the path entry to the original file name. This concatenated name is stored in a static string and made available to the caller by setting **prealName* to point to it if *prealName* is non-NULL and if the open succeeds. If the entire *path* is tried, and still nothing works, then we try each entry in the library path *libpath* next. The static string will be trashed on the next call to this routine. Also, no individual file name is allowed to be more than 200 characters long; excess characters are lost.

PaSubsWD replaces all uses of the working directory in a path by some fixed directory. It returns a pointer to a path that is just like *path*, except that every implicit or explicit use of the working directory (''.'') is replaced by the *newWD* argument. The result is a static array, which will be trashed on the next call to this procedure.

PaEnum is used to call a client procedure with each directory in *path* prepended to the string *file*. The client procedure is of the form *(*func)(name, cdata)*, where *name* is a directory in the path prepended to *file*, and *cdata* is the same as *cdata* passed to *PaEnum*. This client procedure should return 0 normally, or 1 to abort the path enumeration. If a directory in the search path refers to a non-existent user name (using the ''~user'' syntax), we skip that component. *PaEnum* returns 0 if all clients returned 0, or 1 if some client returned 1. If some client returns 1, the enumeration is aborted.

SEE ALSO

magicutils (3)

NAME

runstats – keep track of process time and memory utilization (in libmagicutils.a)

SYNOPSIS

#include <sys/types.h>
#include <sys/times.h>
#include <sys/time.h>
#include runstats.h

char *RunStats(flags, lastt, deltat)
int flags;
struct tms *lastt, *deltat;

char *RunStatsRealTime()

DESCRIPTION

RunStats collects information about a process's utilization of memory and CPU time. Depending on the flags provided, the following information is returned:

RS_TCUM
cumulative user and system time

RS_TINCR
the difference between the current cumulative user and system time and the values stored in the *tms* struct pointed to by *lastt*. This struct is usually the one last passed to *RunStats* when it was last called with **RS_TINCR** as a flag.

RS_MEM
the number of bytes by which the data segment has grown past its initial size.

RunStats returns a pointer to a statically allocated character string of the form ''**[**... *stuff* ...**]**'', where *stuff* contains the information specified by the flags. Times are of the form *mins*:*secsu mins*:*secss*, where the first time is the amount of user CPU time this process has used, and the second time is the amount of system time used. Memory is specified by a string of the form *N***k**, where *N* is the number of kilobytes by which the data segment has grown past its initial size.

If **RS_TINCR** is specified, the parameters *lastt* and *deltat* are set if they are non-NULL. Both point to *tms* structs; the one pointed to by *deltat* is set to the difference between the current user/system time and the time given in the *tms* struct pointed to by *lastt*; the one pointed to by *lastt* is then set to the current user/system time.

RunStatsRealTime reports the real time, both since the first invocation and incremental since the last invocation. It returns a statically allocated string of the form *x***:***xx***.***x x***:***xx***.***x*, where the first number is the amount of elapsed real time since the first call to *RunStatsRealTime*, and the second is the amount of elapsed real time since the latest call.

BUGS

The interfaces to *RunStats* and *RunStatsRealTime* should really be consistent.

SEE ALSO

magicutils (3)

NAME

set – procedures for setting parameters (from strings) and for printing their values.

SYNOPSIS

Void SetNoisyInt(parm,valueS,file)
int *parm;
char *valueS;
FILE *file;

Void SetNoisyBool(parm,valueS,file)
bool *parm;
char *valueS;
FILE *file;

Void SetNoisyDI(parm,valueS,file)
DoubleInt *parm;
char *valueS;
FILE *file;

DESCRIPTION

These procedures interpet a string and set a parameter accordingly. Error messages are printed if the string doesn't make sense, and in any event the final parameter value is printed. If *valueS* is **NULL**, the parameter value is not changed. If *file* is **NULL** the result is printed with *TxPrintf*, Magic's standard print function, otherwise it is printed on the specified file.

SEE ALSO

magicutils (3)

NAME

show – procedure for displaying rects as feedback, for debugging. printing their values.

SYNOPSIS

Void ShowRect(def, r, style)
CellDef *def;
Rect *r;
int style;

DESCRIPTION

Highlights the specified area in the specified cell and the specified style. See the Magic *garouter* module for example uses.

SEE ALSO

magicutils (3)

NAME

stack – procedures for managing stacks in libmagicutils.a

SYNOPSIS

```
#include magic.h
#include stack.h

Stack *StackNew(sincr)
int sincr;

StackFree(stack)
Stack *stack;

ClientData StackPop(stack)
Stack *stack;

ClientData StackLook(stack)
Stack *stack;

StackPush(arg, stack)
ClientData arg;
Stack *stack;

StackEnum(stack, func, cdata)
Stack *stack;
int (*func)(item, i, cdata)
ClientData item, cdata;
int i;

StackCopy(src, dest, copystr)
Stack *src, **dest;
bool copystr;

bool StackEmpty(stack)
Stack *stack;

ClientData STACKPOP(stack)
Stack *stack;

ClientData STACKLOOK(stack)
Stack *stack;

STACKPUSH(arg, stack)
ClientData arg;
Stack *stack;
```

DESCRIPTION

These procedures implement a simple stack mechanism, allowing stacks containing an arbitrary number of one-word elements to be created, manipulated, and destroyed.

StackNew creates and returns a new *Stack*. This stack grows automatically as new items are pushed on it. The number of new elements for which space is added each time the stack grows is specified by *sincr*. When the stack is through being used, *StackFree* frees it.

Elements can be pushed on the stack using *StackPush*. The top of the stack can be viewed without removing it by using *StackLook*, or removed by *StackPop*. Both return the top element from the stack, or NULL if the stack is empty. Fast macro versions exist for each of these functions: *STACKPUSH*, *STACKLOOK*, and *STACKPOP*. To test whether *stack* is empty, one can call *StackEmpty*, which returns **TRUE** if the stack is empty, or **FALSE** if it contains any entries.

StackEnum visits all the elements in *stack* without popping them. It applies *(*func)()* to each element. The arguments to *(*func)()* are *item*, the stack element being visited, *i*, its index on the stack (1 for the top of the stack, increasing as one moves down the stack), and the same *cdata* as was passed to *StackEnum*. If *(*func)()* returns a non-zero value, the enumeration of the stack aborts and *StackEnum* returns 1; otherwise, *StackEnum* returns 0 after visiting all elements in the stack.

StackCopy is used to make a copy of a stack *src*. It leaves **dest* pointing to the copy. If the parameter *copystr* is **TRUE**, then the elements of *src* are interpreted as pointers to NULL-terminated ASCII strings, which are copied into newly allocated memory before the address of the new string is stored in **dest*; otherwise, the elements of *src* are just copied to **dest*.

BUGS

There should be a way of declaring a *Stack* that pushes or pops more than a single word at a time.

SEE ALSO

magicutils (3)

NAME

string – procedures for manipulating strings in libmagicutils.a

SYNOPSIS

```
#include magic.h
#include utils.h

typedef struct { char *d_str; } LookupTable;

int Lookup(str, table)
char *str;
char *table[];

int LookupStruct(str, table, size)
char *str;
LookupTable *table;
int size;

int LookupAny(c, table)
char c;
char *table[];

int LookupFull(name, table)
char *name;
char *table[];

char *StrDup(oldstr, str)
char **oldstr, *str;

bool StrIsWhite(str, commentok)
char *str;
bool commentok;

bool StrIsInt(str)
char *str;

bool Match(pattern, string)
char *pattern, *string;

char *ArgStr(pargc, pargv, argType)
int *pargc;
char ***pargv;
char *argType;
```

DESCRIPTION

This collection of procedures provide a number of useful functions for dealing with strings. *Lookup* searches a table of strings to find one that matches a given string. It's useful mostly for command lookup. The table of strings should be terminated with a NULL string pointer, and the entries should be alphabetical and all lower-case. Any characters following the first white space in an entry are ignored. If *str* is an unambiguous abbreviation for one of the entries in *table*, then the index of the matching entry is returned. If *str* is an abbreviation for more than one entry in table, then -1 is returned. If *str* doesn't match any entry, then -2 is returned. Case differences are ignored.

LookupStruct is a more general version of *Lookup* for dealing with tables of structures whose first word is a string pointer. The *table* argument should be a pointer to an array of such structures, cast as type *(LookupTable *)*. The table should be terminated with an entry whose string pointer is NULL. As in *Lookup*, all entries should contain lower-case strings and should be sorted alphabetically. The *size* parameter gives the size in bytes of each structure in the table.

LookupAny looks up a single character in a table of pointers to strings. The last entry in the string table must be a NULL pointer. The index of the first string in the table containing the indicated character is returned, or -1 if no matching string is found.

LookupFull is like *Lookup*, but does not allow abbreviations. It either returns the index of the entry of *table* matching *str*, or -1 if no match is found. Case is significant, and entries are considered to extend all the way to their trailing NULL byte, instead of being terminated by the first white space as in *Lookup*.

StrDup can be used to replace an old string with a new one, freeing the storage for the old one and allocating sufficient storage for the new one. It returns a pointer to a newly allocated character array just large enough to hold *str* and its trailing NULL byte. This newly allocated array contains a copy of *str*. However, if *str* is NULL, no memory is allocated and we return NULL. If *oldstr* is non-NULL, then if **oldstr* is non-NULL, *StrDup* frees the storage pointed to by **oldstr*. *StrDup* then sets **oldstr* to point to the new array of memory just allocated, or NULL if *str* was NULL.

StrIsWhite returns TRUE if *str* is all white space, or FALSE otherwise. If *commentok* is TRUE, then if the first non-white character in *str* is a pound-sign (''**#**''), *str* is considered to be all white space.

StrIsInt returns TRUE if *str* is a well-formed decimal integer, or FALSE if it isn't.

Match provides a *csh* (1)-like wild-card matching facility. The string *pattern* may contain any of the *csh* wildcard characters: *****, **?**, \, **[**, and **]**. If *pattern* matches *string*, *Match* returns TRUE; otherwise, it returns FALSE.

ArgStr is provided to allow standard processing of command-line arguments that take parameters. It recognizes flag-value pairs of either the form ''*–Xvalue*'' (a single argument string) or ''*–X value*'' (two successive argument strings) in the argument list (**pargc*, **pargv*), incrementing **pargc* and **pargv* by an amount sufficient to step over the flag-value pair. If there are no more arguments remaining in the list, *ArgStr* prints an error message complaining that *argType* is required for the flag **pargv[0]*.

SEE ALSO

magicutils (3)

NAME

cmap – format of .cmap files (color maps)

DESCRIPTION

Color-map files define the mapping between eight-bit color numbers and red, green and blue intensities used for those numbers. They are read by Magic as part of system startup, and also by the **:load** and **:save** commands in color-map windows. Color-map file names usually have the form *x*.*y*.*z*.**cmap***n*, where *x* is a class of technology files, *y* is a class of displays, *z* is a class of monitors, and *n* is a version number (currently **1**). The version number will change in the future if the formap of color-map files ever changes. Normally, *x* and *y* correspond to the corresponding parts of a display styles file. For example, the color map file **mos.7bit.std.cmap1** is used today for most nMOS and CMOS technology files using displays that support at least seven bits of color per pixel and standard-phosphor monitors. It corresponds to the display styles file **mos.7bit.dstyle5**.

Color-map files are stored in ASCII form, with each line containing four decimal integers separated by white space. The first three integers are red, green, and blue intensities, and the fourth field is a color number. For current displays the intensities must be integers between 0 and 255. The color numbers must increase from line to line, and the last line must have a color number of 255. The red, green, and blue intensities on the first line are used for all colors from 0 up to and including the color number on that line. For other lines, the intensities on that line are used for all colors starting one color above the color number on the previous line and continuing up and through the color number on the current line. For example, consider the color map below:

```
255     0     0     2
0       0     255   3
255     255   255   256
```

This color map indicates that colors 0, 1, and 2 are to be red, color 3 is to be blue, and all other colors are to be white.

SEE ALSO

magic (1), dstyle (5)

NAME

displays – Display Configuration File

DESCRIPTION

The interactive graphics programs Caesar, Magic, and Gremlin use two separate terminals: a text terminal from which commands are issued, and a color graphics terminal on which graphical output is displayed. These programs use a **displays** file to associate their text terminal with its corresponding display device.

The **displays** file is an ASCII text file with one line for each text terminal/graphics terminal pair. Each line contains 4 items separated by spaces: the name of the port attached to a text terminal, the name of the port attached to the associated graphics terminal, the phosphor type of the graphics terminal's monitor, and the type of graphics terminal.

An applications program may use the phosphor type to select a color map tuned to the monitor's characteristics. Only the **std** phosphor type is supported at UC Berkeley.

The graphics terminal type specifies the device driver a program should use when communicating with its graphics terminal. Magic supports types **UCB512**, **AED1024**, and **SUN120**. Other programs may recognize different display types. See the manual entry for your specific application for more information.

A sample displays file is:

```
/dev/ttyi1 /dev/ttyi0 std UCB512
/dev/ttyj0 /dev/ttyj1 std UCB512
/dev/ttyjf /dev/ttyhf std UCB512
/dev/ttyhb /dev/ttyhc std UCB512
/dev/ttyhc /dev/ttyhb std UCB512 ®.in -0.5i
```

In this example, **/dev/ttyi1** connects to a text terminal. An associated **UCB512** graphics terminal with standard phosphor is connected to **/dev/ttyi0**.

FILES

Magic uses the displays file ~cad/lib/displays. Gremlin looks in /usr/local/displays.

SEE ALSO

magic(1)

NAME

dlys – format of .dlys files read by the SCALD simulator and timing verifier

DESCRIPTION

The SCALD simulator and timing verifier can accept information about the actual delays of wires in a circuit. This delay information is described in a **.dlys** file, which consists of a sequence of records, one for each electrical net. Each record begins with the signal name for the net (note that this is the SCALD signal name, i.e, the name given by the user to the entire net, and not usually the name of one of the pins in the net), followed by an `=`, then a comma-separated list of the terminals in the net and their associated delay, with the list terminated by a semicolon. The end of the file is marked with a second semicolon.

The elements of the comma-separated list for each net take the form

location **[***min***:***max***]**

where *location* is the full hierarchical SCALD name of the physical pin to which the delay is computed, and *min* and *max* are the best-case and worst-case wire delay in nanoseconds (both are floating-point numbers). The assumption is that only a single driver exists per net, so all delays are computed from this driver. If a net has multiple drivers, then the interpretation of delays is up to the program reading this file (e.g, *min* delays are taken from the fastest driver, *max* from the slowest).

Here is an example **.dlys** file:

```
(APS )ALU STATUS BITS I1<0> =
    (APS MR 3V6 R1 1P )IN#63[ 0.3 : 0.4 ],
    (APS APS 4RI RFC RF )OUT[ 0.5 : 0.7 ];
(APS )ALU STATUS BITS I1<1> =
    (APS APS 4ALUD DCD )AN#12[ 1.4 : 1.6 ],
    (APS APS 4ALUD DCD )AN#8[ 1.1 : 1.3 ],
    (APS APS 4ALUD DCD )AN#9[ 1.1 : 1.3 ],
    (APS APS 4ALUD DCD )AN#10[ 1.1 : 1.3 ],
    (APS APS 4ALUD DCD )AN#11[ 1.1 : 1.3 ],
    (APS MR 3V2 R1 1P )#23  [ 0.6 : 0.8 ],
    (APS MR 3V6 R1 1P )#62  [ 0.3 : 0.4 ],
    (APS APS 4ALUD DCD )    [ 0.4 : 0.6 ],
    (APS APS 4ALUD DCD )#1  [ 0.4 : 0.6 ],
    (APS APS 4ALUD DCD )#2  [ 0.4 : 0.6 ],
    (APS APS 4ALUD DCD )#3  [ 0.4 : 0.6 ],
    (APS APS 4ALUD DCD )#4  [ 0.7 : 0.8 ],
    (APS APS 4ALUD DCD )#5  [ 0.7 : 0.8 ];
;
```

Although it is not good practice, it is possible to omit the actual pin names from the *location* names and only give the path to the part; the example above shows several cases where the final pin name is missing. Since the timing verifier and simulator have the original SCALD netlist available, they are usually able to use the signal name to determine the net, and then use the part's path to identify which pin of the net is meant. This is accurate when a net connects to at most one pin per part; if it connects to more than one pin per part then there is ambiguity over which pin is meant. Usually, though, this ambiguity results in only a small inaccuracy, since the delay to different pins on the same part is usually similar. Also, if delay is capacitive, the delay to all pins in a net will be the same anyway, so there is no inaccuracy.

SEE ALSO

ext2dlys (1), ext (5), sim (5)

BUGS

There should be some way to specify which pins are drivers and which are receivers in a net.

The ability to omit pin names is dangerous; although it usually works it can introduce large inaccuracies when the parts are large compared to the sizes of the wires used to connect them, as might be the case on a silicon PCB.

NAME

dstyle – format of .dstyle files (display styles)

DESCRIPTION

Display styles indicate how to render information on a screen. Each style describes one way of rendering information, for example as a solid area in red or as a dotted outline in purple. Different styles correspond to mask layers, highlights, labels, menus, window borders, and so on. See ''Magic Maintainer's Manual #3: Display Styles, Color Maps, and Glyphs'' for more information on how the styles are used.

Dstyle files usually have names of the form *x*.*y*.**dstyle***n*, where *x* is a class of technologies, *y* is a class of displays, and *n* is a version number (currently **5**). The version number may increase in the future if the format of dstyle files changes. For example, the display style file **mos.7bit.dstyle5** provides all the rendering information for our nMOS and CMOS technologies for color displays with at least 7 bits of color.

Dstyle files are stored in ASCII as a series of lines. Lines beginning with ''#'' are considered to be comments and are ignored. The rest of the lines of the file are divided up into two sections separated by blank lines. There should not be any blank lines within a section.

DISPLAY_STYLES SECTION

The first section begins with a line **display_styles** *planes* where *planes* is the number of bits of color information per pixel on the screen (between 1 and 8). Each line after that describes one display style and contains eight fields separated by white space: *style writeMask color outline fill stipple shortName longName* The meanings of the fields are:

style The number of this style, in decimal. Styles 1 through 64 are used to display mask layers in the edit cell. The style number(s) to use for each mask layer is (are) specified in the technology file. Styles 65-128 are used for displaying mask layers in non-edit cells. If style *x* is used for a mask layer in the edit cell, style *x*+64 is used for the same mask layer in non-edit cells. Styles above 128 are used by the Magic code for various things like menus and highlights. See the file *styles.h* in Magic for how styles above 128 are used. When redisplaying, the styles are drawn in order starting at 1, so the order of styles may affect what appears on the screen.

writeMask
This is an octal number specifying which bit-planes are to be modified when this style is rendered. For example, 1 means only information in bit-plane 0 will be affected, and 377 means all eight bit-planes are affected.

color An octal number specifying the new values to be written into the bit-planes that are modified. This is used along with *writeMask* to determine the new value of each pixel that's being modified: newPixel = (oldPixel & ~writeMask) | (color & writeMask) The red, green, and blue intensities displayed for each pixel are not deterimined directly by the value of the pixel; they come from a color map that maps the eight-bit pixel values into red, green, and blue intensities. Color maps are stored in separate files.

outline If this field is zero, then no outline is drawn. If the field is non-zero, it specifies that outlines are to be drawn around the rectangular areas rendered in this style, and the octal value gives an eight-bit pattern telling how to draw the outline. For example, 377 means to draw a solid line, 252 means to draw a dotted line, 360 specifies long dashes, etc. This field only indicates *which* pixels will be modified: the *writeMask* and *color* fields indicate how the pixels are modified.

fill This is a text string specifying how the areas drawn in this style should be filled. It must have one of the values **solid**, **stipple**, **cross**, **outline**, **grid**. **Solid** means that every pixel in the area is to modified according to *writeMask* and *color*. **Stipple** means that the area should be stippled: the stipple pattern given by *stipple* is used to determine which pixels in the area are to be modified. **Cross** means that an X is drawn in a solid line between the diagonally-opposite corners of the area being rendered. **Outline** means that the area should not be filled at all; only an outline is drawn (if specified by *outline*). **Grid** is a special style used to draw a grid in the line style given by

outline. The styles **cross** and **stipple** may be supplemented with an outline by giving a non-zero *outline* field. The **outline** and **grid** styles don't make sense without an an outline, and **solid** doesn't make sense with an outline (since all the pixels are modified anyway).

stipple Used when *fill* is **stipple** to specify (in decimal) the stipple number to use.

shortName

This is a one-character name for this style. These names are used in the specification of glyphs and also in a few places in the Magic source code. Most styles have no short name; use a ''-'' in this field for them.

longName

A more human-readable name for the style. It's not used at all by Magic.

STIPPLES SECTION

The second section of a dstyle file is separated from the first by a blank line. The first line of the second section must be **stipples** and each additional line specifies one stipple pattern with the syntax *number pattern name Number* is a decimal number used to name the stipple in the *stipple* fields of style lines. *Number* must be no less than 1 and must be no greater than a device-dependent upper limit. Most devices support at least 15 stipple patterns. *Pattern* consists of eight octal numbers, each from 0-377 and separated by white space. The numbers form an 8-by-8 array of bits indicating which pixels are to be modified when the stipple is used. The *name* field is just a human-readable description of the stipple; it isn't used by Magic.

FILES

~cad/lib/magic/sys/mos.7bit.dstyle5

SEE ALSO

magic (1), cmap (5), glyphs (5)

NAME

ext – format of .ext files produced by Magic's hierarchical extractor

DESCRIPTION

Magic's extractor produces a **.ext** file for each cell in a hierarchical design. The **.ext** file for cell *name* is *name***.ext**. This file contains three kinds of information: environmental information (scaling, timestamps, etc), the extracted circuit corresponding to the mask geometry of cell *name*, and the connections between this mask geometry and the subcells of *name*.

A **.ext** file consists of a series of lines, each of which begins with a keyword. The keyword beginning a line determines how the remainder of the line is interpreted. The following set of keywords define the environmental information:

tech *techname*

Identifies the technology of cell *name* as *techname*, e.g, **nmos**, **cmos**.

timestamp *time*

Identifies the time when cell *name* was last modified. The value *time* is the time stored by Unix, i.e, seconds since 00:00 GMT January 1, 1970. Note that this is *not* the time *name* was extracted, but rather the timestamp value stored in the **.mag** file. The incremental extractor compares the timestamp in each **.ext** file with the timestamp in each **.mag** file in a design; if they differ, that cell is re-extracted.

version *version*

Identifies the version of **.ext** format used to write *name***.ext**. The current version is **5.1**.

style *style*

Identifies the style that the cell has been extracted with.

scale *rscale cscale lscale*

Sets the scale to be used in interpreting resistance, capacitance, and linear dimension values in the remainder of the **.ext** file. Each resistance value must be multiplied by *rscale* to give the real resistance in milliohms. Each capacitance value must be multiplied by *cscale* to give the real capacitance in attofarads. Each linear dimension (e.g, width, height, transform coordinates) must be multiplied by *lscale* to give the real linear dimension in centimicrons. Also, each area dimension (e.g, transistor channel area) must be multiplied by *scale*scale* to give the real area in square centimicrons. At most one **scale** line may appear in a **.ext** file. If none appears, all of *rscale*, *cscale*, and *lscale* default to 1.

resistclasses *r1 r2 ...*

Sets the resistance per square for the various resistance classes appearing in the technology file. The values *r1*, *r2*, etc. are in milliohms; they are not scaled by the value of *rscale* specified in the **scale** line above. Each node in a **.ext** file has a perimeter and area for each resistance class; the values *r1*, *r2*, etc. are used to convert these perimeters and areas into actual node resistances. See ''Magic Tutorial #8: Circuit Extraction'' for a description of how resistances are computed from perimeters and areas by the program **ext2sim**.

The following keywords define the circuit formed by the mask information in cell *name*. This circuit is extracted independently of any subcells; its connections to subcells are handled by the keywords in the section after this one.

node *name R C x y type a1 p1 a2 p2 ... aN pN*

Defines an electrical node in *name*. This node is referred to by the name *name* in subsequent **equiv** lines, connections to the terminals of transistors in **fet** lines, and hierarchical connections or adjustments using **merge** or **adjust**. The node has a total capacitance to ground of *C* attofarads, and a lumped resistance of *R* milliohms. For purposes of going back from the node name to the geometry defining the node, *(x, y)* is the coordinate of a point inside the node, and *type* is the layer on which this point appears. The values *a1*, *p1*, ... *aN*, *pN* are the area and perimeter for the

material in each of the resistance classes described by the **resistclasses** line at the beginning of the **.ext** file; these values are used to compute adjusted hierarchical resistances more accurately. **NOTE:** since many analysis tools compute transistor gate capacitance themselves from the transistor's area and perimeter, the capacitance between a node and substrate (GND!) normally does not include the capacitance from transistor gates connected to that node. If the **.sim** file was produced by *ext2sim*(1), check the technology file that was used to produce the original **.ext** files to see whether transistor gate capacitance is included or excluded; see ''Magic Maintainer's Manual #2: The Technology File'' for details.

attr *name xl yl xh yh type text*

One of these lines appears for each label ending in the character ''**@**'' that was attached to geometry in the node *name*. The location of each attribute label (*xl yl xh yh*) and the type of material to which it was attached (*type*) are given along with the text of the label minus the trailing ''**@**'' character (*text*).

equiv *node1 node2*

Defines two node names in cell *name* as being equivalent: *node1* and *node2*. In a collection of node names related by **equiv** lines, exactly one must be defined by a **node** line described above.

fet *type xl yl xh yh area perim sub GATE T1 T2 ...*

Defines a transistor in *name*. The kind of transistor is *type*, a string that comes from the technology file and is intended to have meaning to simulation programs. The coordinates of a square entirely contained in the gate region of the transistor are *(xl, yl)* for its lower-left and *(xh, yh)* for its upper-right. All four coordinates are in the *name*'s coordinate space, and are subject to scaling as described in **scale** above. The gate region of the transistor has area *area* square centimicrons and perimeter *perim* centimicrons. The substrate of the transistor is connected to node *sub*.

The remainder of a **fet** line consists of a series of triples: *GATE, T1, ...* Each describes one of the terminals of the transistor; the first describes the gate, and the remainder describe the transistor's non-gate terminals (e.g, source and drain). Each triple consists of the name of a node connecting to that terminal, a terminal length, and an attribute list. The terminal length is in centimicrons; it is the length of that segment of the channel perimeter connecting to adjacent material, such as polysilicon for the gate or diffusion for a source or drain.

The attribute list is either the single token ''0'', meaning no attributes, or a comma-separated list of strings. The strings in the attribute list come from labels attached to the transistor. Any label ending in the character ''^'' is considered a gate attribute and appears on the gate's attribute list, minus the trailing ''^''. Gate attributes may lie either along the border of a channel or in its interior. Any label ending in the character ''**$**'' is considered a non-gate attribute. It appears on the list of the terminal along which it lies, also minus the trailing ''**$**''. Non-gate attributes may only lie on the border of the channel.

The keywords in this section describe information that is not processed hierarchically: path lengths and accurate resistances that are computed by flattening an entire node and then producing a value for the flattened node.

killnode *node*

During resistance extraction, it is sometimes necessary to break a node up into several smaller nodes. The appearance of a **killnode** line during the processing of a **.ext** file means that all information currently accumulated about *node*, along with all fets that have a terminal connected to *node*, should be thrown out; it will be replaced by information later in the **.ext** file. The order of processing **.ext** files is important in order for this to work properly: children are processed before their parents, so a **killnode** in a parent overrides one in a child.

resist *node1 node2 R*

Defines a resistor of *R* milliohms between the two nodes *node1* and *node2*. Both names are

hierarchical.

distance *name1 name2 dmin dmax*

Gives the distance between two electrical terminals *name1* (a driver) and *name2* (a receiver). Note that these are terminals and not nodes: the names (which are hierarchical label names) are used to specify two different locations on the same electrical node. The two distances, *dmin* and *dmax*, are the lengths (in lambda) of the shortest and longest acyclic paths between the driver and receiver.

The keywords in this last section describe the subcells used by *name*, and how connections are made to and between them.

use *def use-id TRANSFORM*

Specifies that cell *def* with instance identifier *use-id* is a subcell of cell *name*. If cell *def* is arrayed, then *use-id* will be followed by two bracketed subscript ranges of the form: **[***lo***,***hi***,***sep***]**. The first range is for x, and the second for y. The subscripts for a given dimension are *lo* through *hi* inclusive, and the separation between adjacent array elements is *sep* centimicrons.

TRANSFORM is a set of six integers that describe how coordinates in *def* are to be transformed to coordinates in the parent *name*. It is used by *ext2sim*(1) in transforming transistor locations to coordinates in the root of a design. The six integers of *TRANSFORM* (*ta, tb, tc, td, te, tf*) are interpreted as components in the following transformation matrix, by which all coordinates in *def* are post-multiplied to get coordinates in *name*:

ta	*td*	0
tb	*te*	0
tc	*tf*	1

merge *path1 path2 C a1 p1 a2 p2 ... aN pN*

Used to specify a connection between two subcells, or between a subcell and mask information of *name*. Both *path1* and *path2* are hierarchical node names. To refer to a node in cell *name* itself, its pathname is just its node name. To refer to a node in a subcell of *name*, its pathname consists of the *use-id* of the subcell (as it appeared in a **use** line above), followed by a slash (*/*), followed by the node name in the subcell. For example, if *name* contains subcell *sub* with use identifier *sub-id*, and *sub* contains node *n*, the full pathname of node *n* relative to *name* will be *sub-id/n*.

Connections between adjacent elements of an array are represented using a special syntax that takes advantage of the regularity of arrays. A use-id in a path may optionally be followed by a range of the form **[***lo***:***hi***]** (before the following slash). Such a use-id is interpreted as the elements *lo* through *hi* inclusive of a one-dimensional array. An element of a two-dimensional array may be subscripted with two such ranges: first the y range, then the x range.

Whenever one *path* in a **merge** line contains such a subscript range, the other must contain one of comparable size. For example,

merge sub-id[1:4,2:8]/a sub-id[2:5,1:7]/b

is acceptable because the range 1:4 is the same size as 2:5, and the range 2:8 is the same size as 1:7.

When a connection occurs between nodes in different cells, it may be that some resistance and capacitance has been recorded redundantly. For example, polysilicon in one cell may overlap polysilicon in another, so the capacitance to substrate will have been recorded twice. The values *C*, *a1*, *p1*, etc. in a **merge** line provide a way of compensating for such overlap. Each of *a1*, *p1*, etc. (usually negative) are added to the area and perimeter for material of each resistance class to give an adjusted area and perimeter for the aggregate node. The value *C* attofarads (also usually negative) is added to the sum of the capacitances (to substrate) of nodes *path1* and *path2* to give the capacitance of the aggregate node.

cap *node1 node2 C*

Defines a capacitor between the nodes *node1* and *node2*, with capacitance *C*. This construct is used to specify both internodal capacitance within a single cell and between cells.

AUTHOR

Walter Scott

SEE ALSO

ext2sim (1), magic (1)

NAME

glyphs – format of .glyphs files

DESCRIPTION

Glyph files (''.glyph'' extension) are used to store commonly-used bit patterns (glyphs) for Magic. Right now, the bit patterns are used for two purposes in Magic. First, they specify patterns for programmable cursors: each cursor shape (e.g. the arrow used for the wiring tool) is read in as a glyph from a glyph file. Second, glyphs are used by the window manager to represent the icons displayed at the ends of scroll bars. Glyph file names normally have the extension **.glyph**.

Glyph files are stored in ASCII format. Lines beginning with ''#'' are considered to be comments and are ignored. Blank lines are also ignored. The first non-comment line in a glyph file must have the syntax **size** *nGlyphs width height* The *nGlyphs* field must be a number giving the total number of glyphs stored in the file. The *width* and *height* fields give the dimensions of each glyph in pixels. All glyphs in the same file must have the same size.

The **size** line is followed by a description for each of the glyphs. Each glyph consists of *height* lines each containing 2×*width* characters. Each pair of characters corresponds to a bit position in the glyph, with the leftmost pair on the topmost line corresponding to the upper-left pixel in the glyph.

The first character of each pair specifies the color to appear in that pixel. The color is represented as as a single character, which must be the short name of a display style in the current display style file. Some commonly-used characters are **K** for black, **W** for white, and **.** for the background color (when **.** is used in a cursor, it means that that pixel position is transparent: the underlying picture appears through the cursor). See ''Magic Maintainer's Manual #3: Display Styles, Color Maps, and Glyphs'' for more information.

The second character of each pair is normally blank, except for one pixel per glyph which may contain a ''*'' in the second character. The ''*'' is used for programmable cursors to indicate the hot-spot: the pixel corresponding to the ''*'' is the one that the cursor is considered to point to.

For an example of a glyph file, see ~cad/lib/magic/sys/color.glyphs.

SEE ALSO

magic (1), dstyle (5)

NAME

magic – format of **.mag** files read/written by Magic

DESCRIPTION

Magic uses its own internal ASCII format for storing cells in disk files. Each cell *name* is stored in its own file, named *name***.mag**.

The first line in a **.mag** file is the string

magic

to identify this as a Magic file.

The next line is optional and is used to identify the technology in which a cell was designed. If present, it should be of the form

tech *techname*

If absent, the technology defaults to a system-wide standard, currently **nmos**.

The next line is also optional and gives a timestamp for the cell. The line is of the format

timestamp *stamp*

where *stamp* is a number of seconds since 00:00 GMT January 1, 1970 (i.e, the Unix time returned by the library function *time()*). It should be the last time this cell or any of its children changed. The timestamp is used to detect when a child is edited outside the context of its parent (the parent stores the last timestamp it saw for each of its children; see below). When this occurs, the design-rule checker must recheck the entire area of the child for subcell interaction errors. If this field is omitted in a cell, Magic supplies a default value that forces the rechecks.

Next come lines describing the contents of the cell. There are three kinds of groups of lines, describing mask rectangles, subcell uses, and labels. Each group must appear contiguously in the file, but the order between groups is arbitrary.

Each group of mask rectangles is headed with a line of the format

<< *layer* >>

where *layer* is a layername known in the current technology (see the **tech** line above). Each line after this header has the format

rect *xbot ybot xtop ytop*

where *(xbot, ybot)* is the lower-left corner of the rectangle in Magic (lambda) coordinates, and *(xtop, ytop)* is the upper-right corner. Degenerate rectangles are not allowed; *xbot* must be strictly less than *xtop*, and *ybot* strictly less than *ytop*. The smallest legal value of *xbot* or *ybot* is **–67108858**, and the largest legal value for *xtop* or *ytop* is **67108858**. Values that approach these limits (within a factor of 100 or 1000) may cause numerical overflows in Magic even though they are not strictly illegal. We recommend using coordinates around zero as much as possible.

Rectangles should be non-overlapping, although this is not essential. They should also already have been merged into maximal horizontal strips (the neighbor to the right or left of a rectangle should not be of the same type), but this is also not essential.

The second kind of line group describes a single cell use. Each cell use group is of the following form:

use *filename use-id*
array *xlo xhi xsep ylo yhi ysep*
timestamp *stamp*
transform *a b c d e f*
box *xbot ybot xtop ytop*

A group specifies a single instance of the cell named *filename*, with instance-identifier *use-id*. The instance-identifier *use-id* must be unique among all cells used by this **.mag** file. If *use-id* is not specified, a unique one is generated automatically.

The **array** line need only be present if the cell is an array. If so, the X indices run from *xlo* to *xhi* inclusive, with elements being separated from each other in the X dimension by *xsep* lambda. The Y indices run from *ylo* to *yhi* inclusive, with elements being separated from each other in the Y dimension by *ysep* lambda. If *xlo* and *xhi* are equal, *xsep* is ignored; similarly if *ylo* and *yhi* are equal, *ysep* is ignored.

The **timestamp** line is optional; if present, it gives the last time this cell was aware that the child *filename* changed. If there is no **timestamp** line, a timestamp of 0 is assumed. When the subcell is read in, this value is compared to the actual value at the beginning of the child cell. If there is a difference, the ''timestamp mismatch'' message is printed, and Magic rechecks design-rules around the child.

The **transform** line gives the geometric transform from coordinates of the child *filename* into coordinates of the cell being read. The six integers *a*, *b*, *c*, *d*, *e*, and *f* are part of the following transformation matrix, which is used to postmultiply all coordinates in the child *filename* whenever their coordinates in the parent are required:

a	*d*	0
b	*e*	0
c	*f*	1

Finally, **box** gives an estimate of the bounding box of cell *filename* (covering all the elements of the array if an **array** line was present), in coordinates of the cell being read.

The third kind of line group in a **.mag** file is a list of labels. It begins with the line

<< labels >>

and is followed by zero or more lines of the following form:

rlabel *layer xbot ybot xtop ytop position text*

Here *layer* is the name of one of the layers specified in the technology file for this cell. The label is attached to material of this type. *Layer* may be **space**, in which case the label is not considered to be attached to any layer.

Labels are rectangular. The lower-left corner of the label (the part attached to any geometry if *layer* is non-**space**) is at *(xbot, ybot)*, and the upper-right corner at *(xtop, ytop)*. The width of the rectangle or its height may be zero. In fact, most labels in Magic have a lower-left equal to their upper right.

The text of the label, *text*, may be any sequence of characters not including a newline. This text is located at one of nine possible orientations relative to the center of the label's rectangle. *Position* is an integer between 0 and 8, each of which corresponds to a different orientation:

0	center
1	north
2	northeast

3	east
4	southeast
5	south
6	southwest
7	west
8	northwest

A **.mag** file is terminated by the line

<< end >>

Everything following this line is ignored.

Any line beginning with a pound sigh (''**#**'') is considered to be a comment and ignored. Beware, however, that these comments are discarded by Magic when it reads a cell, so if that cell is written again by Magic, the comments will be lost.

NOTE FOR PROGRAMS THAT GENERATE MAGIC FILES

Magic's incremental design rule checker expects that every cell is either completely checked, or contains information to tell the checker which areas of the cell have yet to be examined for design-rule violations. To make sure that the design-rule checker verifies all the material that has been generated for a cell, programs that generate **.mag** files should place the following rectangle in each file:

<< checkpaint >>
rect *xbot ybot xtop ytop*

This rectangle may appear anywhere a list of rectangles is allowed; immediately following the **timestamp** line at the beginning of a **.mag** file is a good place. The coordinates *xbot* etc. should be large enough to completely cover anything in the cell, and must surround all this material by at least one lambda. Be careful, however, not to make this area too ridiculously large. For example, if you use the maximum and minimum legal tile coordinates, it will take the design-rule checker an extremely long time to recheck the area.

SEE ALSO

magic (1)

NAME

net – format of .net files read/written by Magic's netlist editor

DESCRIPTION

Netlist files are read and written by Magic's netlist editor in a very simple ASCII format. The first line contains the characters '' Netlist File'' (the leading blank is important). After that comes a blank line and then the descriptions of one or more nets. Each net contains one or more lines, where each line contains a single terminal name. The nets are separated by blank lines. Any line that is blank or whose first character is blank is considered to be a separator line and the rest of its contents are ignored.

Each terminal name is a path, much like a file path name in Unix. It consists of one or more fields separated by slashes. The last field in the path is the name of a label in a cell. The other fields (if any), are cell instance identifiers that form a path from the edit cell down to the label. The first instance identifier must name a subcell of the edit cell, the second must be a subcell of the first, and so on.

Instance identifiers are unique within their parent cells, so a terminal path selects a unique cell to contain the label. However, the same label may appear multiple times within its cell. When this occurs, Magic assumes that the identical labels identify electrically equivalent terminals; it will choose the closest of them when routing to that terminal. Further, after connecting to one of these terminals Magic may take advantage of the internal wiring connecting them together and route through a cell to complete the net's wiring.

An example netlist file follows below. It contains three distinct nets.

```
 Netlist File

alu/bit_1/cout
alu/bit_2/cin


regcell[21,2]/output
latch[2]/input
 This line starts with a blank, so it's a separator.
opcode_pla/out6
shifter/drivers/shift2
```

SEE ALSO

magic (1)

NAME

sim – format of .sim files read by esim, crystal, etc.

DESCRIPTION

The simulation tools *crystal* (1) and *esim* (1) accept a circuit description in **.sim** format. There is a single **.sim** file for the entire circuit, unlike Magic's *ext* (5) format in which there is a **.ext** file for every cell in a hierarchical design.

A **.sim** file consists of a series of lines, each of which begins with a key letter. The key letter beginning a line determines how the remainder of the line is interpreted. The following are the list of key letters understood.

| units: *s* **tech:** *tech* format: *MIT|LBL|SU*

If present, this must be the first line in the **.sim** file. It identifies the technology of this circuit as *tech* and gives a scale factor for units of linear dimension as *s*. All linear dimensions appearing in the **.sim** file are multiplied by *s* to give centimicrons. The format field signifies the sim variant. MIT and SU are compatible and understood by all tools. LBL is understood only by gemini(1).

type g s d l w x y **g=***gattrs* **s=***sattrs* **d=***dattrs*

Defines a transistor of type *type*. Currently, *type* may be **e** or **d** for NMOS, or **p** or **n** for CMOS. The name of the node to which the gate, source, and drain of the transistor are connected are given by *g*, *s*, and *d* respectively. The length and width of the transistor are *l* and *w*. The next two tokens, *x* and *y*, are optional. If present, they give the location of a point inside the gate region of the transistor. The last three tokens are the attribute lists for the transistor gate, source, and drain. If no attributes are present for a particular terminal, the corresponding attribute list may be absent (i.e, there may be no **g=** field at all). The attribute lists *gattrs*, etc. are comma-separated lists of labels. The label names should not include any spaces, although some tools can accept label names with spaces if they are enclosed in double quotes. **In version 6.4.5 and later** the default format produced by ext2sim is SU. In this format the attribute of the gate starting with S_ is the substrate node of the fet. The attributes of the gate, and source and substrate starting with A_, P_ are the area and perimeter (summed for that node only once) of the source and drain respectively. This addition to the format is backwards compatible.

C *n1 n2 cap*

Defines a capacitor between nodes *n1* and *n2*. The value of the capacitor is *cap* femtofarads. **NOTE:** since many analysis tools compute transistor gate capacitance themselves from the transistor's area and perimeter, the capacitance between a node and substrate (GND!) normally does not include the capacitance from transistor gates connected to that node. If the **.sim** file was produced by *ext2sim* (1), check the technology file that was used to produce the original **.ext** files to see whether transistor gate capacitance is included or excluded; see ''Magic Maintainer's Manual #2: The Technology File'' for details.

R *node res*

Defines the lumped resistance of node *node* to be *res* ohms. This construct is only interpreted by a few programs.

r *node1 node2 res*

Defines an explicit resistor between nodes *node1* and *node2* of resistance *res* ohms. This construct is only interpreted by a few programs.

N *node darea dperim parea pperim marea mperim*

As an alternative to computed capacitances, some tools expect the total perimeter and area of the polysilicon, diffusion, and metal in each node to be reported in the **.sim** file. The **N** construct associates diffusion area *darea* (in square centimicrons) and diffusion perimeter *dperim* (in centimicrons) with node *node*, polysilicon area *parea* and perimeter *pperim*, and metal area *marea* and perimeter *mperim*. *This construct is technology dependent and obsolete.*

A *node attr*

Associates attribute *attr* for node *node*. The string *attr* should contain no blanks.

= *node1 node2*

Each node in a **.sim** file is named implicitly by having it appear in a transistor definition. All node names appearing in a **.sim** file are assumed to be distinct. Some tools, such as *esim* (1), recognize aliases for node names. The **=** construct allows the name *node2* to be defined as an alias for the name *node1*. Aliases defined by means of this construct may not appear anywhere else in the **.sim** file.

SEE ALSO

crystal (1), esim (1), ext2sim (1), sim2spice (1), ext (5)

NAME

vdmpc – format of vdmpc color bitmaps (raster images for printing)

SYNTAX

(none)

DESCRIPTION

Vdmpc color bitmap files are processed by the vdmpc filter, which comes with the Magic VLSI layout editor from Berkeley.

A vdmpc file consists of 5 parts: a 1024 byte header and four bitmaps, one for each of the following colors in order: black, cyan, magenta, and yellow. This format is inspired by the 3000-series Versatec color plotters.

The 1024 byte header starts with a 1-word magic number: 0xA5CF4DFB. Thus, the first 4 bytes of the file are: 0x4D, 0xFB, 0xA5, 0xCF. The second word (four bytes) contain the length of the bitmap in pixels. The third word gives the width of the bitmap. The width must be a multiple of 8. It does not have to match the physical width of the plotter, as it will be clipped or padded as needed. The remaining words of the header are currently unused, and should be set to zero.

The rest of the file contains the bitmaps. There are no separation or formatting characters between the bitmaps -- they follow immediately after the header block and are packed back-to-back. Each bitmap is of size width*height bits, where width and height are defined in the header. Each bitmap consists of a sequence of bits packed into bytes. The bits are ordered from the top-left of the image moving towards the top-right. After a complete scan-line is represented this way, the data for the next lower scan line is represented. Bits are packed into bytes such that the leftmost bit on the physical device is placed into the high-order bit position of the byte.

SEE ALSO

vdmpc(8)

AUTHOR

Bob Mayo

NAME

prleak – aid for debugging programs using malloc/free

SYNOPSIS

prleak [**–a**] [**–d**] [**–l**] [*objfile* [*tracefile*]]

DESCRIPTION

Prleak is a tool for use in debugging programs that make use of Magic's versions of *malloc* and *free*. It examines the trace file produced by special versions of *malloc* and *free* produced when they are compiled with the **–DMALLOCTRACE** flag. The output of prleak is the average allocation size, a list of 'leaky' allocations (blocks still allocated at program exit) if **–l** is specified, a list of duplicate frees (blocks that the program attempted to free after they had already been deallocated) if **–d** is specified, and a list of all calls to malloc and free if **–a** is specified. If no switches are given, the default action is as though **–l** and **–d** were in effect.

For each entry output, both the address of the allocated block and a stack backtrace at the time of the call to *malloc* or *free* are printed. *Prleak* attempts to use the namelist from *objfile* (**a.out** if no file is given) to produce a symbolic backtrace. If no namelist can be found, the backtrace is printed in hex. If *tracefile* is specified, the malloc trace is read from it; otherwise, it is read from the file **malloc.out** in the current directory.

An example output might be as follows:

```
Average allocation size = 12 bytes

------
Leaks:
------

0x11540     [11 bytes]
      at _foo+0x14
      called from ~main+026

0x11556     [14 bytes]
      at _bar+0x50
      called from _foo+0x36
      called from ~main+0x26

--------- ------
Duplicate frees:
--------- ------

0x11556
      at _bar+0x40
      called from _foo+0x36
      called from ~main+0x26
```

FILES

malloc.out

SEE ALSO

ACM SIGPLAN Notices, Vol 17, No 5 (May 1982), the article by Barach and Taenzer.

AUTHOR

Walter Scott

BUGS

Local symbols (beginning with ‘‘~’’) in the backtrace output should be tagged with the source file to which they refer.

NAME

vdmpc – bitmap (raster image) filter for a Color Versatec (3000 series)

SYNTAX

/usr/lib/vdmpc -x *width* **-y** *length* **-n** *login* **-h** *host acnt_file*

DESCRIPTION

The **vdmpc** filter is a modification of the vdmp filter. It works with color Versatec plotters, such as the 3000 series. *Width* is the width of the plotter in pixels, taken from the **px** entry in /etc/printcap. *Length* is the length of a **12 inch** plot, in pixels, taken from the **py** entry in /etc/printcap. 12 inches was chosen because older software uses the foot as the accounting unit for roll-fed paper.

Vdmpc uses the vdmpc file format described in vdmpc(5).

AUTHOR

Modifications, of original vdmp program, by Bob Mayo.

SEE ALSO

lpr(1), vdmpc(5)

Magic Tutorial #1: Getting Started

John Ousterhout
(updated by others, too)

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

This tutorial corresponds to Magic version 6.

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 particular set of facilities in Magic. There is also a set of

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 I. The Magic tutorials, maintenance manuals, and technology manuals.

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 I 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** (or ucbvax!ucbarpa!magic). 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 on a mainframe with a serial-line graphics display. 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
```

Under X10, the layout window will appear in the upper left quadrant of your screen. The X11 server will normally prompt you for the window's position and size. This 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".

3.1. 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 log out and then back in again 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, and 7 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} that the display will support.

The X10 driver only supports monochrome and 7 bit displays.

3.2. Serial-line Displays

If you are running Magic on a mainframe, each station consists of a standard video terminal, called the *text display*, and a color display. You use the keyboard on the text display to type in commands, and Magic uses its screen to log the commands and their results. The color display is used to display one or more portions of the circuit you are designing. You will use a graphics tablet or mouse to point to things on the color display and to invoke some commands. If there is a keyboard attached to the color display (as, for example, with AED512 displays) it is not used except to reset the display. The current version of Magic supports the AED family of displays. Most of the displays are now available with special ROMs in them that provide extra Magic support (talk to your local AED sales rep to make sure you get the UCB ROMs). More displays are being added, so check the Unix man page for the most up-to-date information.

4. 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**

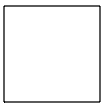
Display is usually the model number of the workstation you are using or the name of your window system. Look in the manual page for a list of valid names, or just guess something. Magic will print out the list of valid names if you guess wrong.

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.

5. 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.

6. 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 6.

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, ^D, 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 **^D** (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 **^D** 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 **^D** macro to try different combinations of paints and erases. If the cursor is over empty space then the **^D** 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 **^D** 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 ‘‘@’’, ‘‘$’’, or ‘‘^’’ is an attribute. There are three different kinds of attributes. Node attributes are those ending with ‘‘@’’; they are associated with particular nodes. Transistor source/drain attributes are those ending in ‘‘$’’; 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 ‘‘^’’ 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: versatec (black-and-white or color), gremlin and pixels (a generalized pixel-file that can be massaged in many ways). The first style is for printers like the black-and-white Versatec family: for these, Magic will output a raster file and spool the file for printing. To plot part of your design, place the box around the part you'd like to plot and type

:plot versatec [*width* [*layers*]]

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 I. 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 I. 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.

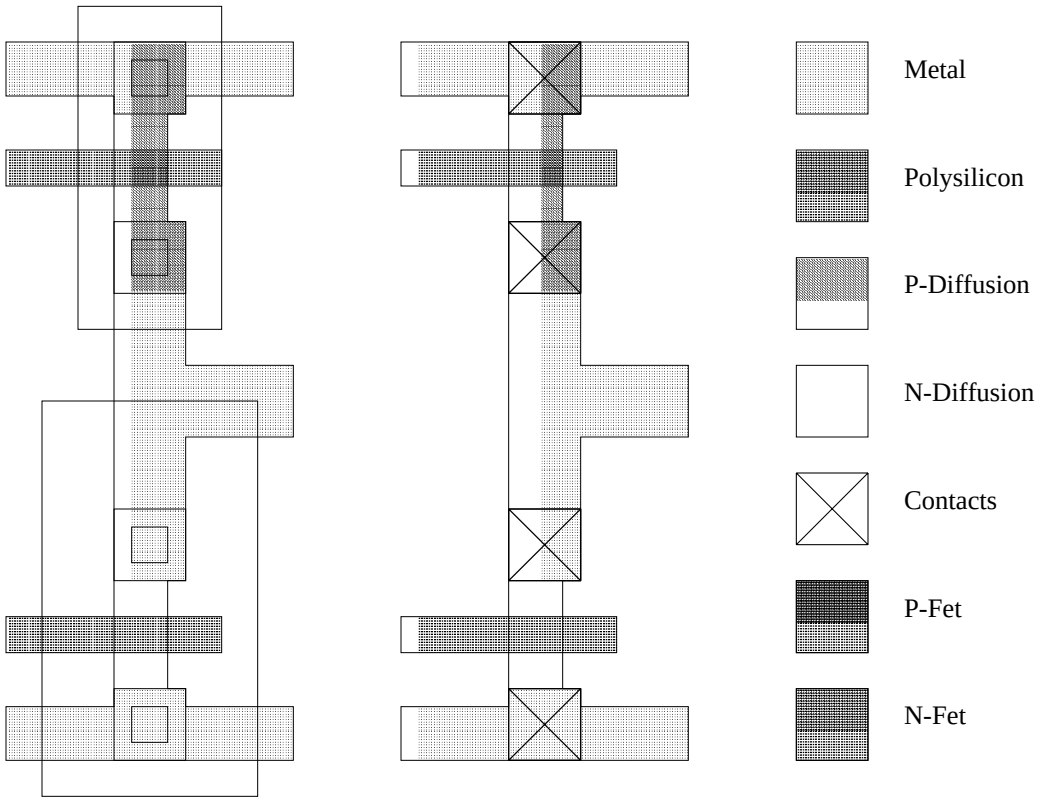


Figure 1. An example of how the logs are used. The figure on the left shows actual mask layers for an CMOS inverter cell, and the figure on the right shows the layers used to represent the cell in Magic.

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.

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 6.

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.

2.1. 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.

2.2. 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.

2.3. 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.

2.4. 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.

2.5. 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.

2.6. 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.

3. 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 6.

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, ^X

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** (''**f**ind 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 **^X** which is equivalent. This causes the internals of that instance of **tut4y** to be displayed. If you type **^X** 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 **^X** 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. **Tut3a** 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 6.

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

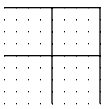
:closewindow

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

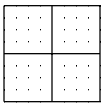
2.2. Resizing and Moving Windows

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 6.

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

Macro 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 ^C) 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 6.

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.

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

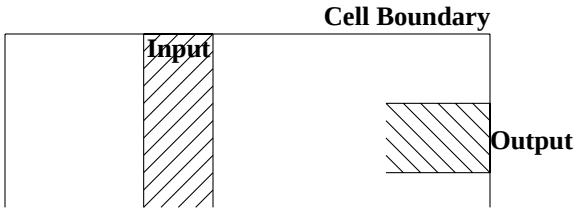


Figure 1. An example of terminal labels. Each terminal should be labeled with a line or rectangle along the edge of the cell.

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.

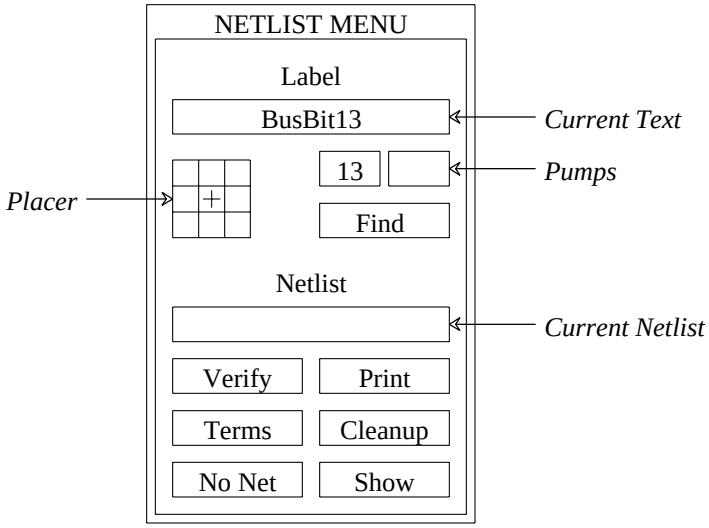


Figure 2. The netlist menu.

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

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 **:cleanup** command)
No Net	Delete selected net (same as **:dnet** command)
Show	Highlight paint connected to material under box (same as typing **:shownet** command)

Table I. A summary of all the netlist menu button actions.

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

Table III. 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.

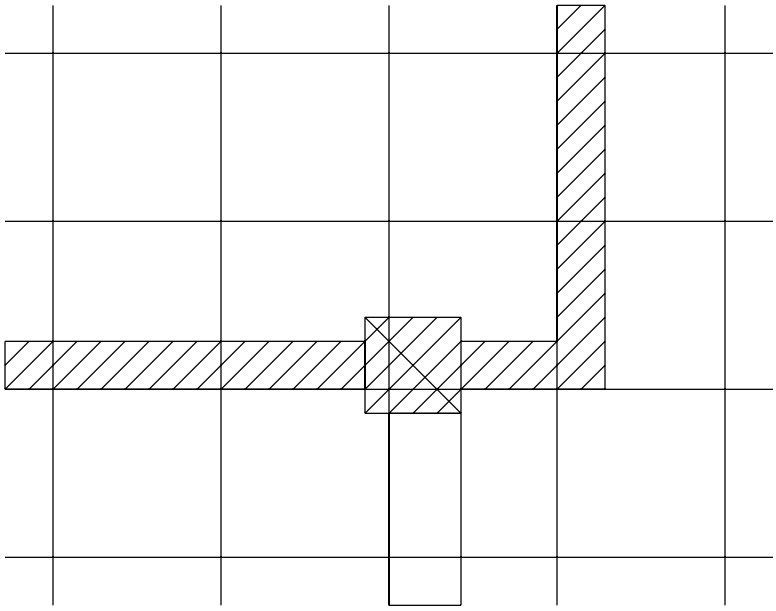


Figure 3. When placing hand routing, it is best to place wires with their left and bottom edges along grid lines, and contacts centered on the wires. In this fashion, the hand routing will block as few routing grid lines as possible.

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
(some updates by other folks, too)

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

This tutorial corresponds to Magic version 6.

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

Programs introduced in this tutorial:

ext2sim

ext2spice

extcheck

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.

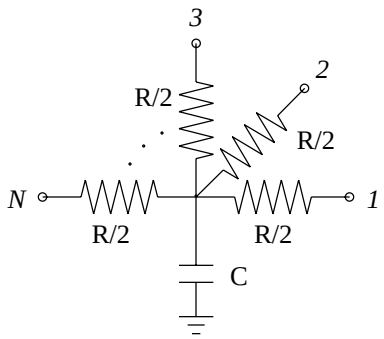


Figure 1. Each node extracted by Magic has a lumped resistance *R* and a lumped capacitance *C* to the substrate. These lumped values can be interpreted as in the diagram above, in which each device connected to the node is attached to one of the points *1*, *2*, ..., *N*.

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 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.

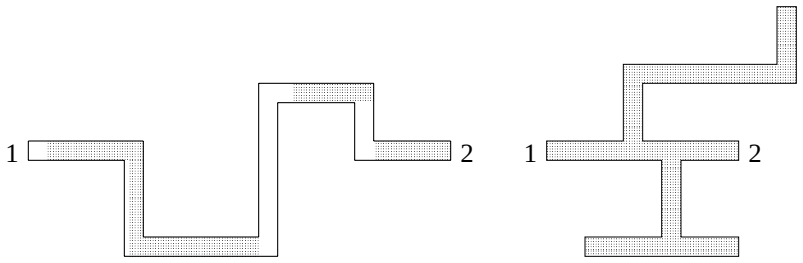


Figure 2. Magic approximates the resistance of a node by assuming that it is a simple wire. The length and width of the wire are estimated from the node’s perimeter and area. (a) For non-branching nodes, this approximation is a good one. (b) The computed resistance for this node is the same as for (a) 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**.

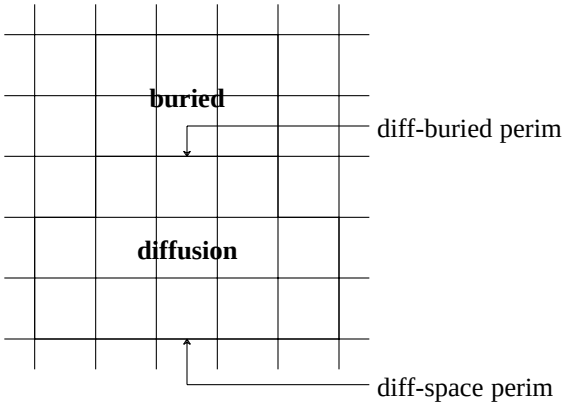


Figure 3. Each type of edge has capacitance to substrate per unit length. Here, the diffusion-space perimeter of 13 units has one value per unit length, and the diffusion-buried perimeter of 3 units another. In addition, each type of material has capacitance per unit area.

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

: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.

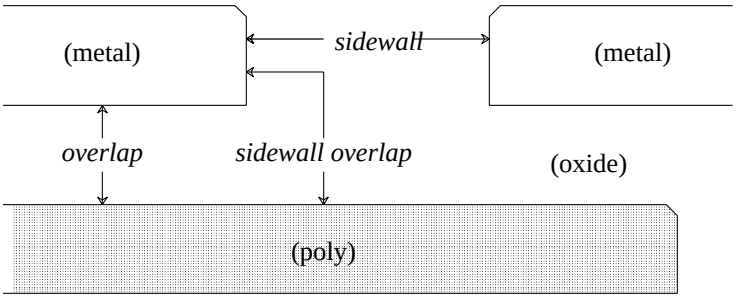


Figure 4. Magic extracts three kinds of internodal coupling capacitance. This figure is a cross-section (side view, not a top view) of a set of masks that shows all three kinds of capacitance. *Overlap* capacitance is parallel-plate capacitance between two different kinds of material when they overlap. *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.

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

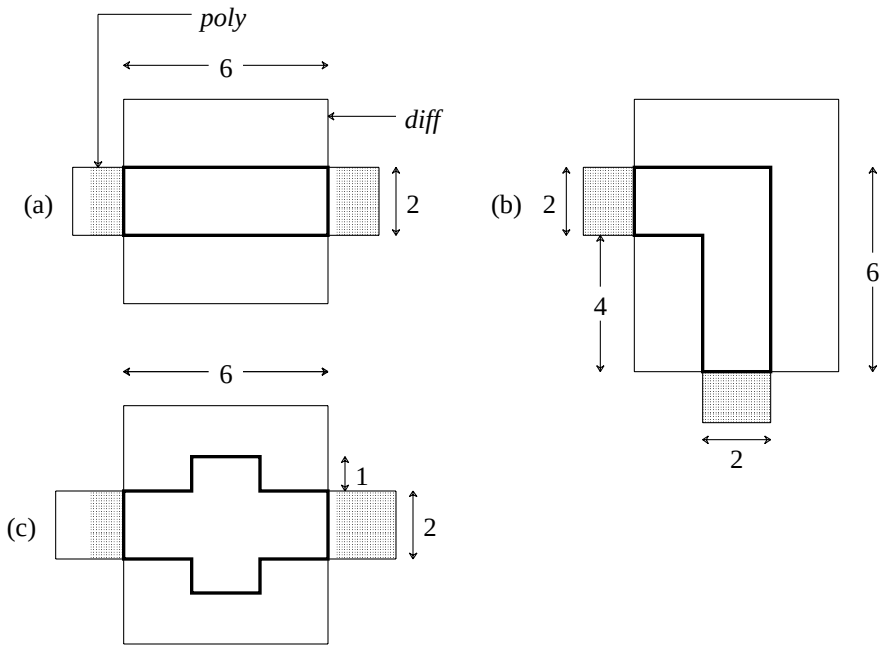


Figure 5.
(a) When transistors are rectangular, it is possible to compute L/W exactly. Here *gateperim* = 4, *srcperim* = 6, *drainperim* = 6, and $L/W = 2/6$. (b) The L/W of non-branching transistors can be approximated. Here *gateperim* = 4, *srcperim* = 6, *drainperim* = 10. By averaging *srcperim* 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.

: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**.

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.

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

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**.

material is connected. This has several advantages: it allows 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 6.

Tutorials to read first:

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

Commands covered in this tutorial:

:calma, :cif

Macros covered 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

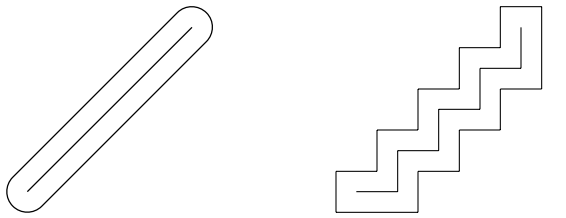
The units used for coordinates in Magic are generally different from those in CIF files. In Magic, most technology files use lambda-based units, where one unit is typically half the minimum feature size. In CIF files, the units are centimicrons (hundredths of a micron). When reading CIF and Calma files, 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.

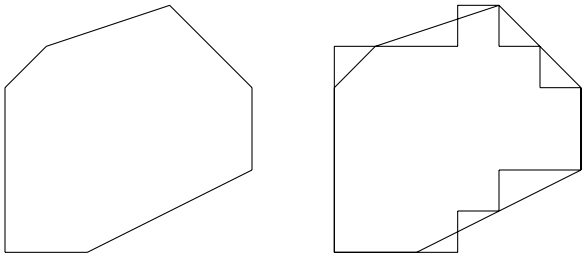
Technology files usually specify geometrical operations such as bloating, shrinking, and-ing, and or-ing to be performed on CIF geometries when they are read into Magic. These geometrical operations are all performed in the CIF coordinate system (centimicrons) so there is no rounding or loss of accuracy 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 6, and Irouter version 0.6.

Tutorials to read first:

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

Commands covered in this tutorial:

:iroute

Macros covered in this tutorial:

Cntl-R, Cntl-N

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.)

:irouter 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 .

15. 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 6.

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/scmos150.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 scmos150.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 **^C** (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 Maintainer's Manual #1: Hints for System Maintainers

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 6.

Tutorials to read first:

All of them.

Commands covered in this tutorial:

:*profile, :*runstats, :*seeflags, :*watch

Macros covered in this tutorial:

None.

1. Introduction

This document provides some information to help would-be Magic maintainers learn about the system. It is not at all complete, and like most infrequently-used documentation, will probably become less and less correct over time as the system evolves but this tutorial doesn't. So, take what you read here with a grain of salt. We believe that everything in this tutorial was up-to-date as of the 1990 DECWRL/Livermore Magic release. Before doing anything to the internals of Magic, you should read at least the first, and perhaps all four, of the papers on Magic that appeared together in the *1984 Design Automation Conference*. In addition, the following portions of magic have their own papers:

extractor	*1985 Design Automation Conference*, page 286.
channel router	*1985 Chapel Hill Conference on VLSI*, page 145.

irouter and mzrouter	*1988 Design Automation Conference*, page 672.
resistance extractor	*1987 Design Automation Conference*, page 570.

2. Installing Magic

If you've received Magic on the 1990 DECWRL/Livermore Magic tape, then it shouldn't take much work to get it running. You should first pick a location for Magic's directory tree. Normally ~cad is chosen, but you might want to pick some other location to start. If you do pick another location, set your shell environment variable **CAD_HOME** to that location and mentally translate the ~cad references in this document to the location you chose. After reading the tape in, there will be a DECstation 3100 binary version of Magic in ~cad/bin and a set of library subdirectories in ~cad/lib/magic. If this isn't so, then at the very least you'll need to get a Magic system library set up in ~cad/lib/magic/sys: this directory contains information like technology files and colormaps and Magic can't run at all without it.

If you're running on a DECstation 3100 you shouldn't need to do anything besides what's mentioned above. Just run X11 and then run Magic. If you are running on some other workstation, you'll need to read the next section on how to make a new binary. The rest of this section concerns serial-line displays, so if you are using any sort of workstation with a built-in display there is no need to read the next couple of paragraphs.

If you're running on a mainframe with a serial-line color display, you'll probably need to do some additional setup. If the display is an AED512 or similar display, it will be attached to the mainframe via an RS232 port. Magic needs to be able to read from this port, and there are two ways to do this. The first is simply to have no login process for that port and have your system administrator change the protection to allow all processes to read from the port and write to it. The second way is to have users log in on the display and run a process that changes the protection of the display. There is a program called Sleeper that we distribute with Magic; if it's run from an AED port it will set everything up so Magic can use the port. Sleeper is clumsy to use, so we recommend that you use the first solution (no login process).

When you're running on mainframes, Magic will need to know which color display port to use from each terminal port. Users can type this information as command-line switches but it's clumsy. To simplify things, Magic checks the file **~cad/lib/displays** when it starts up. The displays file tells which color display port to use for which text terminal port and also tells what kind of display is attached. Once this file is set up, users can run Magic without worrying about the system configuration. See the manual page for *displays* (5).

One last note: if you're running on an AED display, you'll need to set communication switches 3-4-5 to up-down-up.

3. Source Directory Structure

There are approximately 45 source subdirectories in Magic. Most of these consist of modules of source code for the system, for example **database**, **main**, and **utils**. See Section 4 of this document for brief descriptions of what's in each source directory. Besides the source code, the other subdirectories are:

doc	Contains sources for all the documentation, including *man* pages, tutorials, and maintenance manuals. Subdirectories of **doc**, e.g. **doc/scmos**, contain the technology manuals. The Makefile in each directory can be used to run off the documentation. The tutorials, maintenance manuals, and technology manuals all use the Berkeley Grn/Ditroff package, which means that you can't run them off without Grn/Ditroff unless you change the sources.
include	Contains installed (i.e. ''safe'') versions of all the header files (*.h) from all the modules.
lib	Contains installed (i.e. ''safe'') versions of each of the compiled and linked modules (*.o).
installed	Most sites don't use this directory. If you want it to be used, you can modify the script ~**cad/src/magic/:instclean**. If used, it contains one subdirectory for each of the source code directories. Each subdirectory contains ''safe'' versions of the source files for that module. These files correspond to the installed **.o** files in **lib**.
magic	This directory is where the modules of Magic are combined together to form an executable version of the system.
contributed	This directory contains Magic source code or programs that were contributed by other people. None of it has been tested by any member of the Magic team.
cadlib	This is a symbolic link to the directory where Magic stores cell libraries and official installed versions of technology files and color maps. Normally, **cadlib** is a symbolic link to ~**cad/lib/magic**.

Magic is a relatively large system: there are around 575 source files, 250,000 lines of C code. In order to make all of this manageable, we've organized the sources in a two-level structure. Each module has its own subdirectory, and you can make changes to the module and recompile it by working within that subdirectory. In addition to the information in the subdirectory, there is an ''installed'' version of each module, which consists of the files in the **lib**, **include**, and **installed** subdirectories. The installed version of each module is supposed to be stable and reliable. At Berkeley, when a module is changed it is tested carefully without re-installing it, and is only re-installed when it is in good condition. Note that ''installed'' doesn't mean that Magic users see the module; it only means that other Magic maintainers will see it. Each of the makefiles invokes the script **:instclean** to do the module installation. This optionally will copy files to the installed directory.

By keeping modules separate, it's possible for several maintainers to work at once as long as they are modifying different source subdirectories. Each maintainer works with the uninstalled version of a module, and links that with the installed versions of all other modules. Thus, for example, one maintainer can modify **database/DBcell.c** and another can modify **dbwind/DBWundo.c** at the same time.

4. Making Magic

The top-level Makefile (~cad/src/magic/Makefile) provides many options. Before using the Makefile, be sure to set your CAD_HOME shell environment variable to the location of your top-level cad directory (if it is not the standard ~cad).

The most useful Makefile options are:

make config	Configure the Magic system for a particular type of display or operating system. This just runs the :config shell script to set up a couple of files. The curious may examine the script directly. If your configuration isn't handled by this script, then you can use it simply as a guide as to what to do. Much of the configuration is done with compilation flags. See a later section of this manual for a full listing of them.
make magic	Make a version of Magic. All sub-modules are remade, if needed, and then the final magic binary is produced.
make everything	Same as "make magic". Both options make auxilary programs like ext2sim.
make force	Force recompilation. Like a "make everything", except that object files are first removed to force recompilation.
make clean	Delete files that can be remade, such as binaries.
make install	Install the Magic binaries in ~cad (or $CAD_HOME if you have that set).

Putting together a runnable Magic system proceeds in two steps after a source file has been modified. First, the source file is compiled, and all the files in its module are linked together into a single file *xyz***.o**, where *xyz* is the name of the module. Then all of the modules are linked together to form an executable version of Magic. The command **make** in each source directory will compile and link the module locally; **make install** will compile and link it, and also install it in the **include**, **lib**, and **installed** directories. All makefiles are set up to use the compiler flags found in **~cad/src/magic/misc/DFLAGS** and **~cad/src/magic/misc/CFLAGS**. A list of flags appears later in this manual.

The command **make** in the subdirectory **magic** will produce a runnable version of Magic in that directory, using the installed versions of all modules. To work with the uninstalled version of a module, create another subdirectory identical to **magic**, and modify the Makefile so that it uses uninstalled versions of the relevant modules. For example, the Magic team uses subdirectories **hamachitest**, **mayotest**, **mhatest**, **oustertest**, and **wsstest** that we use to test new versions of modules before installing them. If you want to remake the entire system, type ''make magic'' in the top-level directory (**~cad/src/magic**).

One last thing -- there are some customizations you may want to make to Magic. If magic core dumps, it sends mail to somebody. That somebody is set in the MAIL_COMMAND definition in the file misc/niceabort.c. You may want to change it. Also in the same file is CRASHDIR which says where core dumps should be placed. Paths used by magic are located in misc/paths.h. You shouldn't need to change them,

though, because the CAD_HOME shell environment variable controls what Magic uses for the location of ~cad.

5. Summary of Magic Modules

This section contains brief summaries of what is in each of the Magic source subdirectories.

calma Contains code to read and write Calma Stream-format files. It uses many of the procedures in the **cif** module.

cif Contains code to process the CIF sections of technology files, and to generate CIF files from Magic.

cmwind Contains code to implement special windows for editing color maps.

commands The procedures in this module contain the top-level command routines for layout commands (commands that are valid in all windows are handled in the **windows** module). These routines generally just parse the commands, check for errors, and call other routines to carry out the actions.

database This is the largest and most important Magic module. It implements the hierarchical corner-stitched database, and reads and writes Magic files.

dbwind Provides display functions specific to layout windows, including managing the box, redisplaying layout, and displaying highlights and feedback.

debug There's not much in this module, just a few routines used for debugging purposes.

drc This module contains the incremental design-rule checker. It contains code to read the **drc** sections of technology files, record areas to be rechecked, and recheck those areas in a hierarchical fashion.

ext2sim The **ext2sim** directory isn't part of Magic itself. It's a self-contained program that flattens the hierarchical **.ext** files generated by Magic's extractor into a single file in **.sim** format. See the manual page **ext2sim (1)**.

ext2spice This is another self-contained program. It converts **.ext** files into single file in spice format. See the manual page **ext2spice (1)**.

extflat Contains code that is used by the **extract** module and the **ext2...** programs. The module produces a library that is linked in with the above programs.

extract Contains code to read the **extract** sections of technology files, and to generate hierarchical circuit descriptions (**.ext** files) from Magic layouts.

fsleeper	Like **ext2sim**, this directory is a self-contained program that allows a graphics terminal attached to one machine to be used with Magic running on a different machine. See the manual page **fsleeper (1)**.
garouter	Contains the gate array router from Lawrence Livermore National Labs.
gcr	Contains the channel router, which is an extension of Rivest's greedy router that can handle switchboxes and obstacles in the channels.
graphics	This is the lowest-level graphics module. It contains driver routines for X11 as well as less-used drivers for the AED family of displays and for Sun Windows. The code here does basic clipping and drawing. If you want to make Magic run on a new kind of display, this is the only module that should have to change.
grouter	The files in this module implement the global router, which computes the sequence of channels that each net is to pass through.
irouter	Contains the interactive router written by Michael Arnold at Lawrence Livermore National Labs. This router allows the user to route nets interactively, using special hint layers to control the routing.
macros	Implements simple keyboard macros.
magicusage	Like **ext2sim**, this is also a self-contained program. It searches through a layout to find all the files that are used in it. See **magicusage** (1).
main	This module contains the main program for Magic, which parses command-line parameters, initializes the world, and then transfers control to **textio**.
misc	A few small things that didn't belong anyplace else.
mpack	Contains routines that implement the Tpack tile-packing interface using the Magic database. (not supported)
mzrouter	Contains maze routing routines that are used by the irouter and garouter modules.
net2ir	Contains a program to convert a netlist into irouter commands.
netlist	Netlist manipulation routines.
netmenu	Implements netlists and the special netlist-editing windows.
parser	Contains the code that parses command lines into arguments.
plot	The internals of the **:plot** command.
plow	This module contains the code to support the **:plow** and **:straighten** commands.

prleak Also not part of Magic itself. Prleak is a self-contained program intended for use in debugging Magic's memory allocator. It analyzes a trace of mallocs/frees to look for memory leaks. See the manual page **prleak (8)** for information on what the program does.

resis Resis is a module that does better resistance extraction via the :extresis command. Courtesy of Don Stark of Stanford.

router Contains the top-level routing code, including procedures to read the router sections of technology files, chop free space up into channels, analyze obstacles, and paint back the results produced by the channel router.

select This module contains files that manage the selection. The routines here provide facilities for making a selection, enumerating what's in the selection, and manipulating the selection in several ways, such as moving it or copying it.

signals Handles signals such as the interrupt key and control-Z.

sim Provides an interactive interface to the simulator rsim. Courtesy of Mike Chow of Stanford.

tech This module contains the top-level technology file reading code, and the current technology files. The code does little except to read technology file lines, parse them into arguments, and pass them off to clients in other modules (such as **drc** or **database**).

textio The top-level command interpreter. This module grabs commands from the keyboard or mouse and sends them to the window module for processing. Also provides routines for message and error printout, and to manage the prompt on the screen.

tiles Implements basic corner-stitched tile planes. This module was separated from **database** in order to allow other clients to use tile planes without using the other database facilities too.

undo The **undo** module provides the overall framework for undo and redo operations, in that it stores lists of actions. However, all the specific actions are managed by clients such as **database** or **netmenu**.

utils This module implements a whole bunch of utility procedures, including a geometry package for dealing with rectangles and points and transformations, a heap package, a hash table package, a stack package, a revised memory allocator, and lots of other stuff.

windows This is the overall window manager. It keeps track of windows and calls clients (like **dbwind** and **cmwind**) to process window-specific operations such as redisplaying or processing commands. Commands that are valid in all windows, such as resizing or moving windows, are implemented here.

wiring The files in this directory implement the **:wire** command. There are routines to select wiring material, add wire legs, and place contacts.

6. Portability Issues

Magic runs on a variety of machines. Running "make config" in the top-level source directory sets the compiletime options. If you are porting Magic, you should modify the configuration section at the end of file "misc/magic.h" to suit your machine, by testing compiler flags. No changes should be made that would hamper Magic's operation on other machines.

7. Compilation Switches

Over the years Magic has acquired a number of compilation switches. While it's undesirable to have so many, it seems unavoidable since people use Magic on such a wide variety of machines. The file ~cad/src/magic/misc/DFLAGS should contain the compile switches that you wish to use at your site. All makefiles for Magic reference the common DFLAGS file. The switches in this release are shown below.

These flags are normally setup by running the "make config" script in ~cad/src/magic. Some of them are turned on in "magic.h" when a particular machine configuration is detected.

7.0.1. Machine/OS Compiletime Options

The following switches should be defined automatically by the compiler.

vax
: For VAX machines.

mips
: For mips processors, such as the DECstation 3100.

MIPSEL
: For little-endian mips processors, such as the DECstation 3100.

MIPSEB
: For big-endian mips processors.

pyramid
: For Pyramid machines.

sun
: For Sun machines.

mc68000
: For machines which have a version of the 68000 as the processor.

sparc
: Sparc-based machines.

lint
: Used to bypass things that lint complains about. Don't turn this on. Lint turns it on itself.

If needed, you should put the following switches in the DFLAGS file.

macII
: For the MacII.

SUNVIEW
: Used when including Magic's SunView graphics drivers.

SUN120
: For the Sun120 machine.

BSD4_2
: Used in the utils module to patch around a broken version of flsbuf() that is needed in the VAX version of Unix 4.2 BSD systems. This is rarely needed, since almost all version of Unix now have this bug fixed.

FASYNC
: Hack for some versions of Sun2 software (old stuff).

NO_VARARGS
: Hack for machines without a VARARGS package.

SYSV
: For Unix System V.

Flags defined, if needed, in "magic.h" based on other flags.

BIG_ENDIAN
: Indicates big endian byte ordering is being used.

LITTLE_ENDIAN
: Indicates little endian byte ordering is being used.

NEED_MONCNTL
: Hack for machines without a moncontrol procedure.

NEED_VFPRINTF
: Hack for machines without a vfprintf procedure.

SIG_RETURNS_INT
: Defined in magic.h for systems that expect a signal handler to return an integer rather than a void.

7.0.2. Graphics Driver Compiletime Options

X10
: Used in the graphics module for the X10 driver.

X11
: Used in the graphics module for the X11 driver.

OLD_R2_FONTS
: Used when X11 is release 2. Magic normally assumes release 3.

AED
: Used in the graphics module when compiling for AED displays.

GTCO
: Used in the graphics module when using a GTCO bitpad with an AED display.

7.0.3. Compiletime Options for Module Inclusion

NO_CALMA
: Flag to eliminate the calma module, to reduce the size of Magic.

NO_CIF
: Flag to eliminate the cif module, to reduce the size of Magic.

NO_EXT
: Flag to eliminate the ext module, to reduce the size of Magic.

NO_PLOT
: Flag to eliminate the plot module, to reduce the size of Magic.

NO_ROUTE
: Flag to eliminate the router modules, to reduce the size of Magic.

NO_SIM
: Flag to eliminate the sim module, to reduce the size of Magic.

7.0.4. Debugging Compiletime Options

CELLDEBUG
: Debugging flag for the database module.

COUNTWIDTHCALLS
: Debugging flag for the plow module.

DEBUGWIDTH
: Debugging flag for the plow module.

DRCRULESHISTO
: Debugging/tuning flag for the drc module.

FREEDEBUG
: Memory allocation debugging flag.

MALLOCMEASURE
: Memory allocation debugging flag.

MALLOCTRACE
: Memory allocation debugging flag.

NOMACROS
: Memory allocation debugging flag.

PAINTDEBUG
: Debugging flag for the database painting routines.

PARANOID
: Flag to enable consistency checking. With a system the complexity of Magic, you should always leave this flag turned on.

8. Technology and Other Support Files

Besides the source code files, there are a number of other files that must be managed by Magic maintainers, including color maps, technology files, and other stuff. Below is a listing of those files and where they are located.

8.1. Technology Files

See ''Magic Maintainer's Manual #2: The Technology File'' for information on the contents of technology files. The sources for technology files are contained in the sub-directory **tech**, in files like **scmos.tech** and **nmos.tech**. The technology files that Magic actually uses at runtime are kept in the directory **cadlib/sys**; **make install** in **tech** will copy the sources to **cadlib/sys**. Technology file formats have evolved rapidly during Magic's life, so we use version numbers to allow multiple formats of technology files to exist at once. The installed versions of technology files have names like **nmos.tech26**, where **26** is a version number. The current version is defined in the Makefile for **tech**, and should be incremented if you ever change the format of technology files; if you install a new format without changing the version number, pre-existing versions of Magic won't be able to read the files. After incrementing the version number, you'll also have to re-make the **tech** module since the version number is referenced by the code that reads the files.

8.2. Display Styles

The display style file sources are contained in the source directory **graphics**. See ''Magic Maintainer's Manual #3: The Display Style and Glyph Files'' and the manual page *dstyle*(5) for a description of their contents. **Make install** in **graphics** will copy the files to **cadlib/sys**, which is where Magic looks for them when it executes.

8.3. Glyph Files

Glyph files are described in Maintainer's Manual #3 and the manual page *glyphs*(5); they define patterns that appear in the cursor. The sources for glyph files appear in two places: some of them are in **graphics**, in files like **color.glyphs**, and some others are defined in **windows/window*XX*.glyphs**. When you **make install** in those directories, the glyphs are copied to **cadlib/sys**, which is where Magic looks for them when it executes.

8.4. Color Maps

The color map sources are also contained in the source directory **graphics**. Color maps have names like **mos.7bit.std.cmap**, where **mos** is the name of the technology style to which the color map applies, **7bit** is the display style, and **std** is a type of monitor. If monitors have radically different phosphors, they may require different color maps to achieve the same affects. Right now we only support the **std** kind of monitor. When Magic executes, it looks for color maps in **cadlib/sys**; **make install** in **graphics** will copy them there. Although color map files are textual, you shouldn't edit them by hand; use Magic's color map editing window instead.

9. New Display Drivers

The most common kind of change that will be made to Magic is probably to adapt it for new kinds of color displays. Each display driver contains a standard collection of procedures to perform basic functions such as placing text, drawing filled rectangles, or changing the shape of the cursor. A table (defined in **graphics/grMain.c**) holds the addresses of the routines for the current display driver. At initialization time this table is

filled in with the addresses of the routines for the particular display being used. All graphics calls pass through the table.

If you have a display other than an AED or an X11 workstation, the first thing you should do is check the directory **contributed**. Each of the subdirectories in **contributed** contains additional code that was contributed by some site outside of Berkeley. Each subdirectory should contain a file named **ReadMe** (or something similar), which explains how to install and use the code. If you have any troubles with one of the these drivers, you'll have to contact the people that contributed the driver; we here at Berkeley don't know anything about them. If the contributors wish to be contacted, they will identify themselves in the **ReadMe** file.

If you need to build a new display driver, we recommend starting with the routines for either the AED (all the files in **graphics** with names like **grAed1.c**), or the Sun (names like **grSunW1.c**). For stand-alone displays, the AED routines are probably the easiest to work from; for integrated workstations with pre-existing window packages, the Sun routines may be easiest. Copy the files into a new set for your display, change the names of the routines, and modify them to perform the equivalent functions on your display. Write an initialization routine like **aedSetDisplay**, and add information to the display type tables in **graphics/grMain.c**. At this point you should be all set. There shouldn't be any need to modify anything outside of the graphics module.

10. Debugging and Wizard Commands

Magic seems to work fine under the latest version of dbx, such as the dbx found on the DECstation 3100. The Makefiles are set up to compile all files with the **-g** switch, which creates debugging information in *dbx*'s format.

Because of the size of Magic and the way Unix handles debugging symbols, it's slow to compile a complete version of Magic with debugging information for everything, and the executable file ends up being enormous. To solve this problem the Makefiles are set up to strip off debugging information before installing. Thus, you have to link with uninstalled versions to get debugging information. In most cases, debugging information is only needed for a few modules at a time, namely the modules you're currently modifying. The database module is set up to install with debugging symbols, since it seems to be involved in almost all debugging.

If you try to use older versions of *dbx*, you'll discover that Magic has too many procedures for the default table sizes; *dbx* runs out of space and dies. The solution is either to recompile *dbx* with larger tables or throw away pieces of Magic to reduce the number of procedures (we recommend the first alternative).

There are a number of commands that we implemented in Magic to assist in debugging. These commands are called *wizard commands*, and aren't visible to normal Magic users. They all start with ''*****''. To get terse online help for the wizard commands, type **:help wizard** to Magic. The wizard commands aren't documented very well. Some of the more useful ones are:

***watch** *plane*

This causes Magic to display on the screen the corner-stitched tile structure for one of the planes of the edit cell. For example, ***watch subcell** will display the structure

of the subcell tile plane, including the address of the record for each tile and the values of its corner stitches. Without this command it would have been virtually impossible to debug the database module.

***profile on|off**

If you're using the Unix profiling tools to figure out where the cycles are going, this command can be used to turn profiling off for everything except the particular operation you want to measure. This command doesn't work on many systems, because the operating system doesn't support selective enabling and disabling of profiling.

***runstats**

This command prints out the CPU time usage since the last invocation of this command, and also the total since starting Magic.

***seeflags** *flag*

If you're working on the router, this command allows you to see the various channel router flags by displaying them as feedback areas. The cursor should first be placed over the channel whose flags you want to see.

Magic Maintainer's Manual #2: The Technology File

Walter Scott

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

John Ousterhout

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

(Updated by various folks.)

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

Tutorials to read first:

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

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

Commands covered in this tutorial:

:*watch

Macros covered in this tutorial:

none

Changes since Magic version 4:

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

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

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

Support for extractor extensions (see Tutorial #8)

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

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

1. Introduction

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

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

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

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

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

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

**width allDiff 2 **
''Diffusion width must be at least 2''

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

2. Tech section

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

tech
scmos
end

Table 1. **Tech** section

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

3. Planes, types, and contact sections

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

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

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

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

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

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

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

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

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

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

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

Table 2. **Planes** section

graphic hints.

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

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

Table 3. **Types** section

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

plane names

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

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

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

Table 4. Built-in Magic types.

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

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

current limit.

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

Table 5. **Contact** section

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

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

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

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

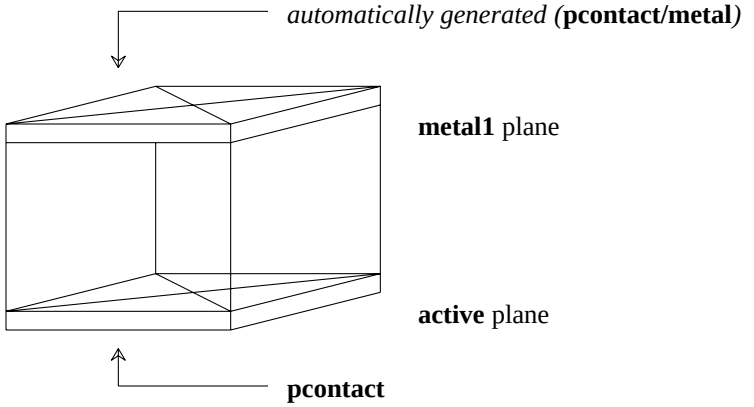


Figure 1. A different tile type is used to represent a contact on each plane that it connects. Here, a contact between poly on the **active** plane and metal1 on the **metal1** plane is stored as two tile types. One, **pcontact**, is specified in the technology file as residing on the **active** plane;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

contact pm12c polysilicon metal1 metal2

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

contact wndc pwell ndiffusion metal1
contact nm12c ndiffusion metal1 metal2

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

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

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

4. Specifying Type-lists

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

poly,ndiff,pcontact,ndc

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

0

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

~(pcontact,ndc)

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

~(pcontact,ndc)/active,metal1

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

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

5. Styles section

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

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

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

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

6. Compose section

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

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

Table 7. **Compose** section

material *PAINT*.

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

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

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

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

compose ntransistor poly ndiff

Sometimes, not all of the ''component'' layers of a type are layers known to magic. As an example, in the **nmos** technology, there are two types of transistors: **enhancement-fet** and **depletion-fet**. Although both contain polysilicon and diffusion, depletion-fet can be thought of as also containing implant, which is not a tile type. So

while we can't construct depletion-fet by painting poly and then diffusion, we'd still like it to behave as though it contained both materials. Painting poly or diffusion over a depletion-fet should not change it, and erasing either poly or diffusion should give the other. These semantics are the result of the following rule:

decompose dfet poly diff

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

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

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

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

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

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

paint pdiffusion pwell ndiffusion

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

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

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

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

completely determined, so the third pass can generate the default rules for all contact types, and the fourth pass can modify these as per any **compose**, etc. rules for contacts.

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

Table 8. **Connect** section

7. Connect section

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

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

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

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

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

8. Cifoutput section

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

layers.

8.1. CIF styles

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

style *name*

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

8.2. Scaling

Each style must contain a line of the form

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

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

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

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

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

stepsize *stepsize*

where *stepsize* is in Magic units. For example, if you plan to generate CIF for designs

that will typically be 100,000 Magic units on a side, it might make sense for *stepsize* to be 10000 or more.

8.3. Layer descriptions

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

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

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

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

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

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

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

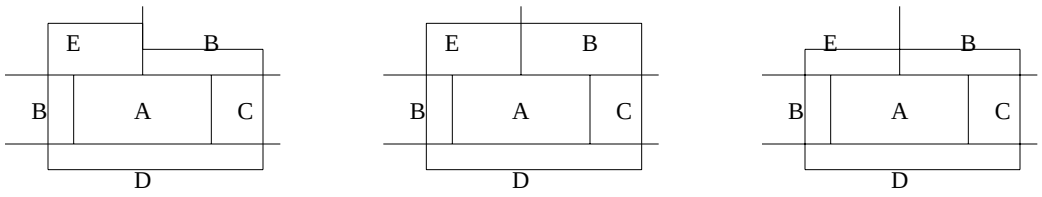


Figure 2. The three different forms of **bloat** behave slightly differently when two different bloat distances apply along the same side of a tile. In each of the above examples, the CIF that would be generated is shown in bold outline. If **bloat-or** is specified, a jagged edge may be generated, as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

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

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

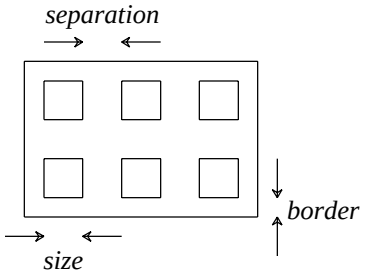


Figure 3. The **squares** operator chops each tile up into squares, as determined by the *border*, *size*, and *separation* parameters. In the example, the bold lines show the CIF that would be generated by a **squares** operation. The squares of material are always centered so that the borders on opposite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

The last geometric operation is called **squares**. It examines each tile on the CIF plane, and replaces that tile with one or more squares of material. Each square is *size* CIF units across, and squares are separated by *separation* units. A border of at least *border* units is left around the edge of the original tile, if possible. This operation is used to generate contact vias, as in Figure 3. If only one argument is given in the **squares**

statement, then *separation* defaults to *size* and *border* defaults to *size*/2. If a tile doesn't hold an integral number of squares, extra space is left around the edges of the tile and the squares are centered in the tile. If the tile is so small that not even a single square can fit and still leave enough border, then the border is reduced. If a square won't fit in the tile, even with no border, then no material is generated. The **squares** operation must be used with some care, in conjunction with the design rules. For example, if there are several adjacent skinny tiles, there may not be enough room in any of the tiles for a square, so no material will be generated at all. Whenever you use the **squares** operator, you should use design rules to prohibit adjacent contact tiles, and you should always use the **no_overlap** rule to prevent unpleasant hierarchical interactions. The problems with hierarchy are discussed in Section 8.6 below, and design rules are discussed in Section 10.

8.4. Labels

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

labels *Magiclayers*

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

8.5. Calma (GDS II Stream format) layers

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

calma *calmaNumber calmaType*

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

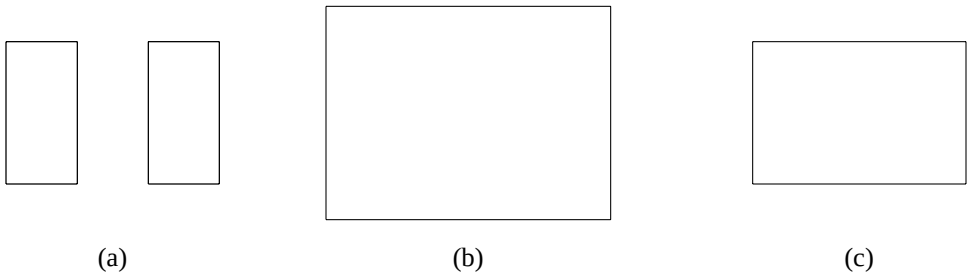


Figure 4. If the operator **grow 100** is applied to the shapes in (a), the merged shape in (b) results. If the operator **shrink 100** is applied to (b), the result is (c). However, if the two original shapes in (a) belong to different cells, and if CIF is generated separately in each cell, the result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

8.6. Hierarchy

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

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

$$Op(A \cup B) \supseteq Op(A) \cup Op(B)$$

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

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

9. Cifinput section

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

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

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

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

style *name*

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

scalefactor *centimicrons*

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

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

layer *magicLayer* [*layers*]

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

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

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

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

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

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

labels *layers*

means that the current Magic layer is to receive all CIF labels on *layers*. This is actually just an initial layer assignment for the labels. Once a CIF cell has been read in, Magic scans the label list and re-assigns labels if necessary. In the example of Table 10, if a label is attached to the CIF layer CPG then it will be assigned to the Magic layer **poly**. However, the polysilicon may actually be part of a poly-metal contact, which is Magic

layer **pcontact**. After all the mask information has been processed, Magic checks the material underneath the layer, and adjusts the label's layer to match that material (**pcontact** in this case). This is the same as what would happen if a designer painted **poly** over an area, attached a label to the material, then painted **pcontact** over the area.

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

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

ignore *cifLayers*

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

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

calma *cifLayer calmaLayers calmaTypes*

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

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

10. Mzrouter section

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

10.1. Styles

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

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

Table 11. Mzrouter section for the scmos technology.

lines pertaining to the irouter should be preceded by

style irouter

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

style garouter

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

10.2. Layers

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

layer *type hCost vCost jogCost hintCost*

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

10.3. Contacts

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

contact *type routeLayer1 routeLayer2 cost*

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

10.4. Notactive

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

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

10.5. Search

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

search *rate width penalty*

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

10.6. Width

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

width *route-type width*

Here *width* is a positive integer.

10.7. Spacing

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

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

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

11. Drc section

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

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

Table 12a. Abbreviations for sets of tile types.

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

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

11.1. Width rules

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

width *type-list width error*

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

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

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

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

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

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

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

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

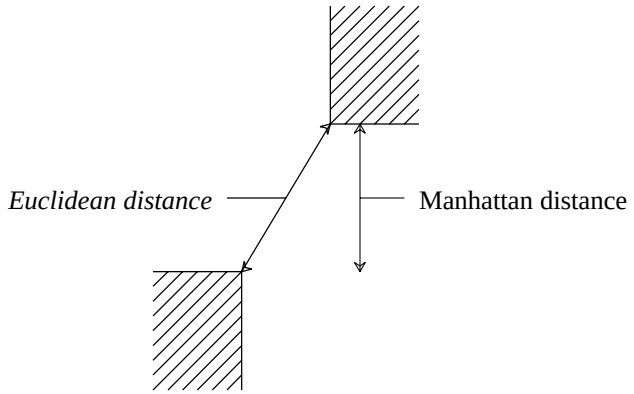


Figure 5. For design rule checking, the Manhattan distance between two horizontally or vertically aligned points is just the normal Euclidean distance. If they are not so aligned, then the Manhattan distance is the length of the longest side of the right triangle forming the diagonal line between the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

11.2. Spacing rules

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

spacing *types1 types2 distance flavor error*

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

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

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

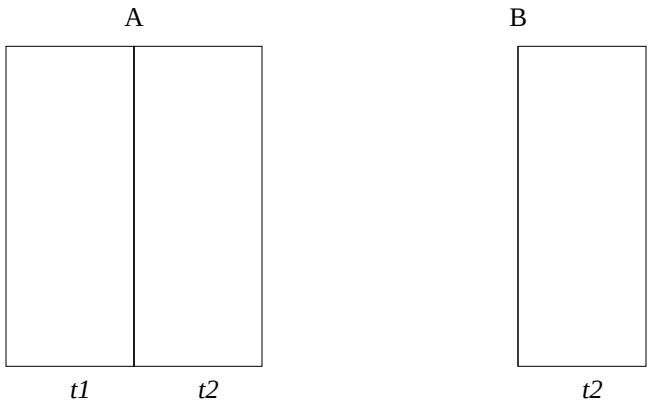


Figure 6. The **touching_ok** rules cancels spacing checks if the material is touching. This means that even distant material won't be checked for spacing. If the rule applied at edge A is a touching_ok rule between material t1 and t2, then no check will be made between the t1 material and the t2 material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

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

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

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

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

11.3. Edge rules

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

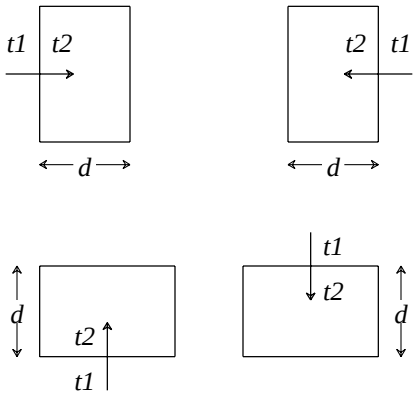


Figure 7. Design rules are applied at the edges between tiles in the same plane. A rule is specified in terms of type *t1* and type *t2*, the materials on either side of the edge. Each rule may be applied in any of four directions, as shown by the arrows. The simplest rules require that only certain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

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

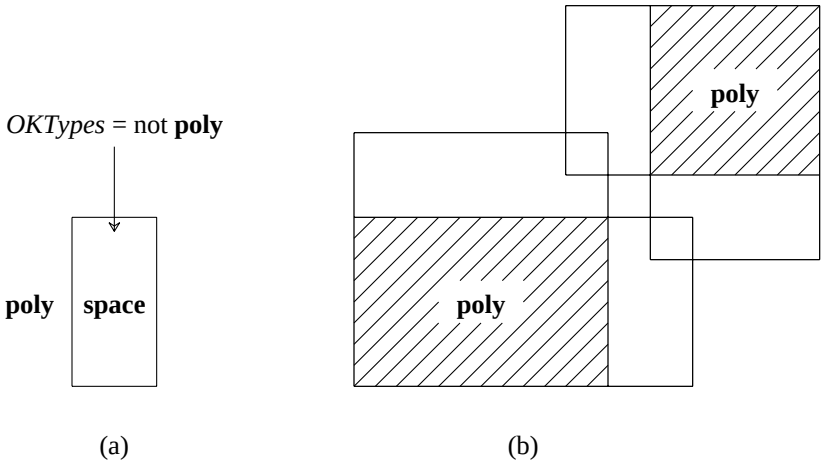


Figure 8. If only the simple rules from Figure 7 are used, errors may go unnoticed in corner regions. For example, the polysilicon spacing rule in (a) will fail to detect the error in (b).

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

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

Both *types1* and *types2* are type-lists. An edge rule is generated for each pair consisting of a type from *types1* and a type from *types2*. All the types in *types1*, *types2*, and *cornerTypes* must lie on a single plane. See Figure 9 for an example edge rule. It is sometimes

useful to specify a null list, i.e., **0**, for *OKTypes* or *CornerTypes*. Null *OKTypes* means no edges between *types1* and *types2* are OK. Null *CornerTypes* means no corner extensions are to be checked (corner extensions are explained below).

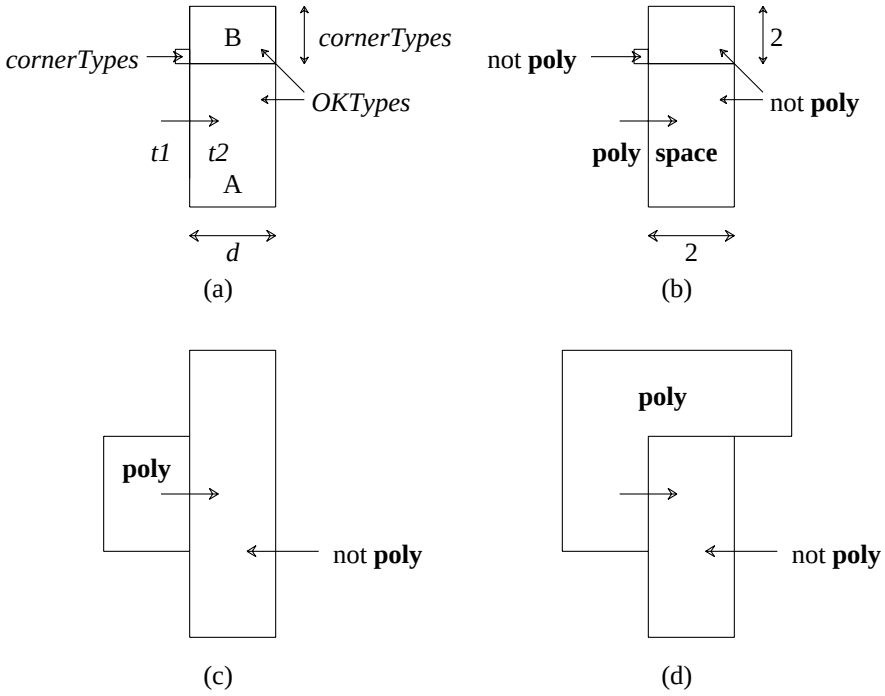


Figure 9. The complete design rule format is illustrated in (a). Whenever an edge has *type1* on its left side and *type2* on its right side, the area A is checked to be sure that only *OKTypes* are present. If the material just above and to the left of the edge is one of *cornerTypes*, then area B is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

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

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

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

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

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

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

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

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

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

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

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

11.4. Overlap Rules

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

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

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

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

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

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

11.5. Background checker step size

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

stepsize *stepsize*

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

12. Extract section

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

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

style *stylename*

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

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

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

areacap *types C*

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

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

perimcap *intypes outtypes C*

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

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

resist *types R*

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

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

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

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

Table 14. **Extract** section

contact *types size R*

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

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

fetresist *fettypes region R*

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

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

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

overlap *toptypes bottomtypes cap* [*shieldtypes*]

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

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

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

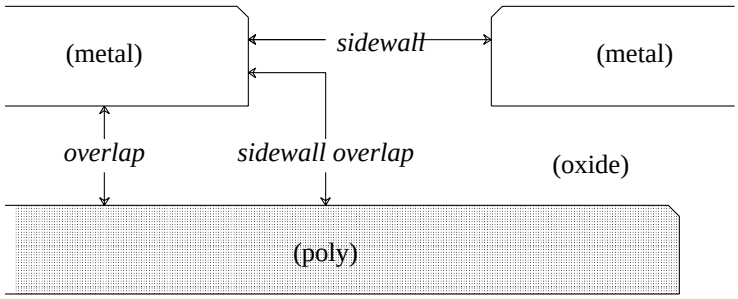


Figure 10. Magic extracts three kinds of internodal coupling capacitance. This figure is a side view of a set of masks that shows all three kinds of capacitance. *Overlap* capacitance is parallel-plate capacitance between two different kinds of material when they overlap. *Parallel wire* capacitance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

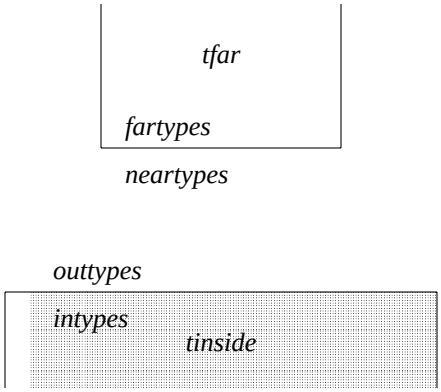


Figure 11. Parallel wire capacitance is between pairs of edges. The capacitance applies between the tiles *tinside* and *tfar* above, where *tinside*'s type is one of *intypes*, and *tfar*'s type is one of *fartypes*.

sidewall *intypes outtypes neartypes fartypes cap*

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

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

sidehalo *distance*

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

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

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

sideoverlap *intypes outtypes ovtypes cap*

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

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

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

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

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

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

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

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

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

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

In the above, *c* is the number of attofarads per unit capacitance appearing in the **.ext** files, *l* is the number of centimicrons per unit length, and *r* is the number of milliohms per unit

resistance. All three must be integers; *r* should divide evenly all the resistance-per-square values specified as part of **resist** lines, and *c* should divide evenly all the capacitance-per-unit values.

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

step *step*

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

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

Table 15. **Wiring** section

13. Wiring section

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

contact *type minSize layer1 surround1 layer2 surround2*

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

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

14. Router section

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

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

Table 16. **Router** section

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

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

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

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

contacts *contactType size* [*surround1 surround2*]

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

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

gridspacing *distance*

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

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

Table 17. **Plowing** section

15. Plowing section

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

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

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

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

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

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

16. Plot section

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

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

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

style *styleName*

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

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

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

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

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

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

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

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

type-list color pat0 pat1 ... pat15

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

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

17. Installing a Technology File

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

other files via the **#include** mechanism are possible.

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

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

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

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

Magic Maintainer's Manual #3:

Display Styles, Color Maps, and Glyphs

Robert N. Mayo
John Ousterhout

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

This tutorial corresponds to Magic version 6.

Tutorials to read first:

All of them.

Commands covered in this tutorial:

none

Macros covered in this tutorial:

none

1. Introduction

This document gives overall information about the files that tell Magic how to display information on the screen. There are three types of files that contain display information: display styles files, color-map files, and glyph files.

2. Display Styles

Display styles files describe how to draw rectangular areas and text. A single file contains a large number of display styles. Each display style contains two kinds of information: a) how to modify pixels (which bits of the pixel should be changed and what their new value(s) should be); and b) which pixels to modify. Part b) consists of things like ''fill the entire area,'' or ''modify only those pixels in the area that are given by a particular stipple pattern,'' or ''draw a dashed-line around the area's outline.'' In the case of text, ''which pixels to modify'' is determined by the font for the text, which is not part

of the display style, so the display style information for this is ignored. See the manual page **dstyle (5)** for details on the format of display styles files.

Display styles are designed to take into account both the characteristics of certain technologies and the characteristics of certain displays. For example, a bipolar process may require information to be displayed very differently than a MOS process, and a black-and-white display will be used much differently than a color display. Thus there can be many different display styles files, each corresponding to a particular class of technologies and a class of displays. The names of styles files reflect these classes: each display styles file has a name of the form *x*.*y*.**dstyle5**, where *x* is the technology class (given by the **styletype** line in the **styles** section of the technology file), and *y* is the class of display. Each display driver knows its display class; the driver initialization routine sets an internal Magic variable with the display class to use. Right now we have two display styles files: **mos.7bit.dstyle5** and **mos.bw.dstyle5**. Both files contain enough different styles to handle a variety of MOS processes, including both nMOS and CMOS (hence the **mos** field). **Mos.7bit.dstyle5** is designed for color displays with at least seven bits of color per pixel, while **mos.bw.dstyle5** is for black-and-white displays (stipple patterns are used instead of colors).

3. Color Maps

The display styles file tells how to modify pixels, but this doesn't completely specify the color that will be displayed on the screen (unless the screen is black-and-white). For color displays, the pixel values are used to index into a *color map*, which contains the red, green, and blue intensity values to use for each pixel value. The values for color maps are stored in color-map files and can be edited using the color-map-editing window in Magic. See **cmap (5)** for details on the format of color-map files.

Each display styles file uses a separate color map. Unfortunately, some monitors have slightly different phosphors than others; this will result in different colors if the same intensity values are used for them. To compensate for monitor differences, Magic supports multiple color maps for each display style, depending on the monitor being used. The monitor type can be specified with the **-m** command line switch to Magic, with **std** as the default. Color-map files have names of the form *x*.*y*.*z*.**cmap1**, where *x* and *y* have the same meaning as for display styles and *z* is the monitor type. Over the last few years monitor phosphors appear to have standardized quite a bit, so almost all monitors now work well with the **std** monitor type. The color map **mos.7bit.std.cmap1** is the standard one used at Berkeley.

4. Transparent and Opaque Layers

One of the key decisions in defining a set of display styles for a color display is how to use the bits of a pixel (this section doesn't apply to black-and-white displays). One option is to use a separate bit of each pixel (called a *bit plane*) for each mask layer. The advantage of this is that each possible combination of layer overlaps results in a different pixel value, and hence a different color (if you wish). Thus, for example, if metal and poly are represented with different bit planes, poly-without-metal, metal-without-poly, poly-and-metal, and neither-poly-nor-metal will each cause a different value to be stored in the pixel. A different color can be used to display each of these combinations. Typically, the colors are chosen to present an illusion of transparency: the poly-and-metal

color is chosen to make it appear as if metal were a transparent colored foil placed on top of poly. You can see this effect if you paint polysilicon, metal1, and metal2 on top of each other in our standard technologies.

The problem with transparent layers is that they require many bits per pixel. Most color displays don't have enough planes to use a different one for each mask layer. Another option is to use a group of planes together. For example, three bits of a pixel can be used to store seven mask layers plus background, with each mask layer corresponding to one of the combinations of the three bits. The problem with this scheme is that there is no way to represent overlaps: where there is an overlap, one of the layers must be displayed at the expense of the others. We call this scheme an *opaque* one since when it is used it appears as if each layer is an opaque foil, with the foils lying on top of each other in some priority order. This makes it harder to see what's going on when there are several mask layers in an area.

The display styles files we've designed for Magic use a combination of these techniques to get as much transparency as possible. For example, our **mos.7bit.dstyle5** file uses three bits of the pixel in an opaque scheme to represent polysilicon, diffusion, and various combinations of them such as transistors. Two additional bits are used, one each, for the two metal layers, so they are transparent with respect to each other and the poly-diff combinations. Thus, although only one poly-diff combination can appear at each point, it's possible to see the overlaps between each of these combinations and each combination of metal1 and metal2. Furthermore, all of these styles are overridden if the sixth bit of the pixel is set. In this case the low order five bits no longer correspond to mask layers; they are used for opaque layers for things like labels and cell bounding boxes, and override any mask information. Thus, for example, when metal1 is displayed it only affects one bit plane, but when labels are displayed, the entire low-order six bits of the pixel are modified. It's important that the opaque layers like labels are drawn after the transparent things that they blot out; this is guaranteed by giving them higher style numbers in the display styles files.

Finally, the seventh bit of the pixel is used for highlights like the box and the selection. All 64 entries in the color map corresponding to pixel values with this bit set contain the same value, namely pure white. This makes the highlights appear opaque with respect to everything else. However, since they have their own bit plane which is completely independent of anything else, they can be drawn and erased without having to redraw any of the mask information underneath. This is why the box can be moved relatively quickly. On the other hand, if Magic erases a label it must redraw all the mask information in the area because the label shared pixel bits with the mask information.

Thus, the scheme we've been using for Magic is a hierarchical combination of transparent and opaque layers. This scheme is defined almost entirely by the styles file, so you can try other schemes if you wish. However, you're likely to have problems if you try anything too radically different; we haven't tried any schemes but the one currently being used so there are probably some code dependencies on it.

For more information on transparent and opaque layers, see the paper ''The User Interface and Implementation of an IC Layout Editor,'' which appeared in *IEEE Transactions on CAD* in July 1984.

5. Glyphs

Glyphs are small rectangular bit patterns that are used in two places in Magic. The primary use for glyphs is for programmable cursors, such as the shapes that show you which corner of the box you're moving and the various tools described in Tutorial #3. Each programmable cursor is stored as a glyph describing the pattern to be displayed in the cursor. The second use of glyphs is by the window package: the little arrow icons appearing at the ends of scroll bars are stored as glyphs, as is the zoom box in the lower-left corner of the window. We may eventually use glyphs in a menu interface (but don't hold your breath).

Glyphs are stored in ASCII glyph files, each of which can hold one or more glyph patterns. Each glyph is represented as a pattern of characters representing the pixels in the glyph. Each character selects a display style from the current display styles file; the display style indicates the color to use for that pixel. See the manual page **glyphs (5)** for details on the syntax of glyphs files.

The window glyphs are stored in files of the form **windows***XX***.glyphs**. The *XX* indicates how wide the glyphs are, and is set by the graphics driver for a particular display. We started out with a **windows7.glyphs** and a **windows11.glyphs**. Since then, display resolution has increased greatly so we have also created a **windows14.glyphs** and a **windows22.glyphs**. The positions of the various glyphs in these files is important, and is defined in the **window** module of Magic.

Programmable cursors are stored in files named *x***.glyphs**, where *x* is determined by the device driver for the display. Displays capable of supporting full-color cursors use **color.glyphs**; displays that can only support monochrome cursors used **bw.glyphs**. The order of the various glyphs in these files is important. It is defined by the files **styles.h** in the **misc** module of Magic.

Magic Maintainer's Manual #4:
Using Magic Under X Windows

Don Stark

Computer Systems Laboratory
Stanford, University
Stanford, CA 94305

This tutorial corresponds to Magic version 6.

Tutorials and man pages to read first:

Magic Tutorial #1: Getting Started
X(1)

Commands covered in this tutorial:

none

Macros covered in this tutorial:

none

1. Introduction

This document provides information on Magic's X drivers that may be of help to system maintainers.

2. Compiling the Correct X Driver for your system.

Unfortunately, it is not possible to link with both the X10 and X11 libraries, so you will have to compile Magic differently depending on the version of X that you are running.

2.1. Compiling for X11

1. Add the flag -DX11 to misc/DFLAGS
2. Add -lX11 to magic/LIBS
3. Change the SRCS line in graphics/Makefile to ${BASE_SRCS} ${X11_SRCS}
4. Change the OBJS line to ${BASE_OBJS} ${X11_OBJS}
5. Change the POBJS line to ${BASE_POBJS} ${X11_POBJS}
6. Change the HELPER_SRCS line ${X11HELPER_SRCS}
7. Change the HELPER_SRCS line ${X11HELPER_PROG}
8. Compile the module graphics.o
9. Relink magic

2.2. Compiling for X10

1. Add the flag -DX10 to misc/DFLAGS
2. Add -lX10 to magic/LIBS
3. Change the SRCS line in graphics/Makefile to ${BASE_SRCS} ${X10_SRCS}
4. Change the OBJS line to ${BASE_SRCS} ${X10_OBJS}
5. Change the POBJS line to ${BASE_SRCS} ${X10_POBJS}
6. Change the HELPER_SRCS line ${X10HELPER_SRCS}
7. Change the HELPER_SRCS line ${X10HELPER_PROG}
8. Compile the module graphics.o
9. Relink magic

3. Troubleshooting the X Drivers

The following is a list of problems sometimes encountered in running Magic under X and some suggestions about how to get around the problem.

3.1. X11 Driver

Fonts — We have tried to pick a set of fonts that most machines running X11 Revision 3 will have, but there is nothing to guarantee that a given machine will have a font. If you're getting "unable to load font" messages, you will need to change the fonts that Magic uses. The simplest way to do this is to specify them in your .Xdefaults file as described in section 2.1. To change the default values that Magic uses, change the "fontnames" array in the file grX11su3.c of the graphics module. The program *xlsfonts* will tell you what fonts are available on your machine.

Strange Color Effects — Magic often co-exists rather uneasily with other X applications because it is picky about which colors it is allocated. If possible, it tries to allocate the colors it requires out of the display's default colormap because this perturbs other applications the least. If this fails, however, Magic makes its own colormap. When this colormap gets installed is a function of the window manager; most window managers install it when the cursor is in the magic window. Unfortunately, there is no way to guarantee that the window manager installs the magic colormap correctly; if you get erratic colormap behavior, try using a lower number

of planes or reducing the number of colors that other applications use.

When magic's colormap is being used, other windows may change color, possibly to some unusable combination such as black on black or white on white. This problem can sometimes be ameliorated by changing the constants X_COLORMAP_BASE and X_COLORMAP_RESERVED in grX11su2.c; a more complete description of what these constants do is included in that file. Values for these constants that are incompatible with your machine will sometimes generate Xerrors in XQueryColors.

Failure to prompt user for window position

Whether or not the designer is prompted for a window's location is dependent on the window manager. Certain window managers, notably *twm*, do not always do this.

3.2. X10 Driver

In general, the Version 10 driver is less reliable than the X11 one. If you have the choice, you are better off running under X11.

grX2.GrXSetCMap: Failed to get color cells

Magic gives this error when it can't get sufficient colors to run. This can be caused by running Magic on a machine with an insufficient number of planes (8 planes are generally required to run a 7 bit dstyles file), or by having too many colors already used by other applications. Try using only black and white xterms, xclocks, etc., and see if the problem goes away.

Couldn´t get 7 planes; allocating by color

Certain X10 servers, most notably the VaxstationII-GPX, allocate colors in such a way that Magic can never get the 7 color planes that it wants. When this happens, Magic instead allocates 128 colors. This is better than nothing, but not by much; strange colors often result when layers overlap.

4. Acknowledgments

Many people share the credit (and the blame) for the Magic X drivers. The original X10 port was done by Mark Linton and Doug Pan at Stanford University. Walter Scott and Eric Lunow of Lawrence Livermore National Laboratories modified the driver and the windows module so that magic windows act like normal X windows. Meanwhile, Dave Durfee and Markus G. Wloka of Brown University improved the reliability of the Stanford X10 driver and added support for a variable number of planes. Marco Papa of USC converted the Brown X10 driver to X11. Concurrently, someone at the University of Washington converted the Stanford X10 driver to X11. The X11 driver in this distribution is predominantly a merge of the UW driver with the multiwindow features of the LLNL driver. Some of the ideas for supporting differing plane counts were borrowed from the USC/Brown work. Thanks to the Digital Equipment Corporation Western

Research Laboratory (DECWRL) for use of their computer facilities, and to Mike Chow of Apple Computer for the Macintosh II-specific changes.

Magic Technology Manual #1: NMOS

John Ousterhout

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

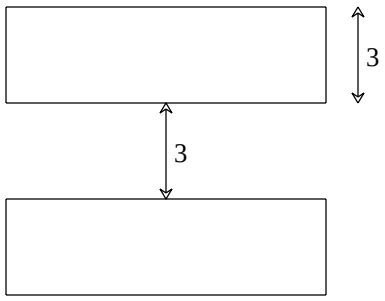
(Warning: Process details often change. Contact MOSIS or your fab line to verify information in this document.)

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

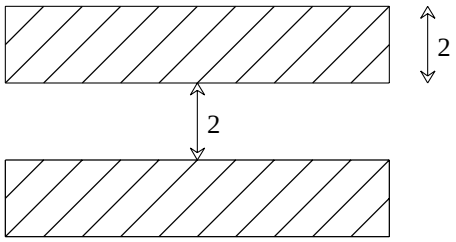
2.1. Metal



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

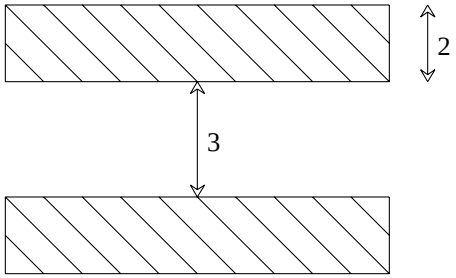
separated from other metal by at least 3 units.

2.2. Polysilicon



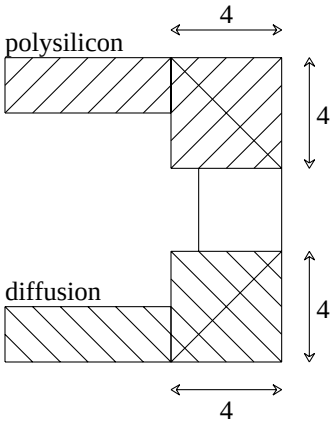
Polysilicon is drawn in red, and can be referred to in Magic as either **polysilicon** or **red**. It has a minimum width of 2 units and a minimum spacing of 2 units.

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

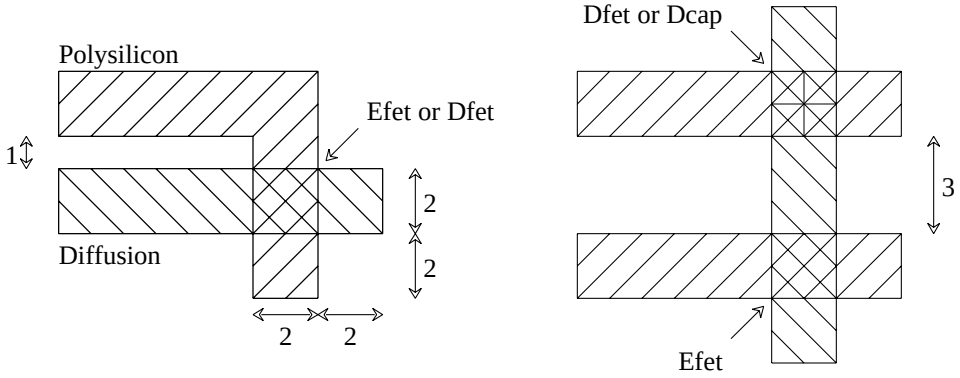


Contacts between metal and polysilicon, and between metal and diffusion, have similar forms. Poly-metal contacts can be referred to as **pmc** or **poly_metal_contact**; they are drawn to look like metal running on top of poly, with an ‘‘X’’ over the area of the contact. Diffusion-metal contacts are 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.

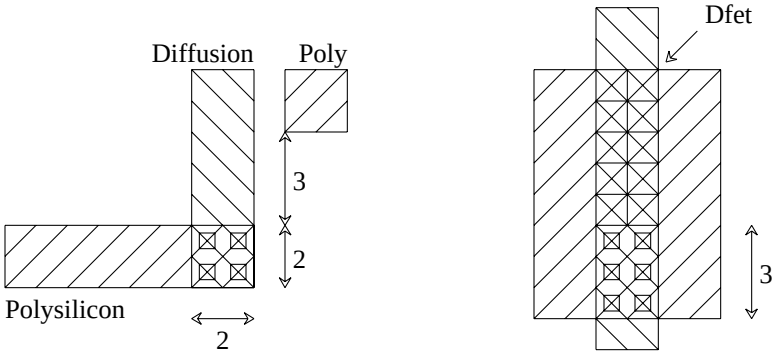
An additional kind of contact, called **glass_contact**, is used to generate holes in the overglass layer for use in bonding to pads. This layer is drawn as gray stripes over blue, and includes both metal and the overglass hole.

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.

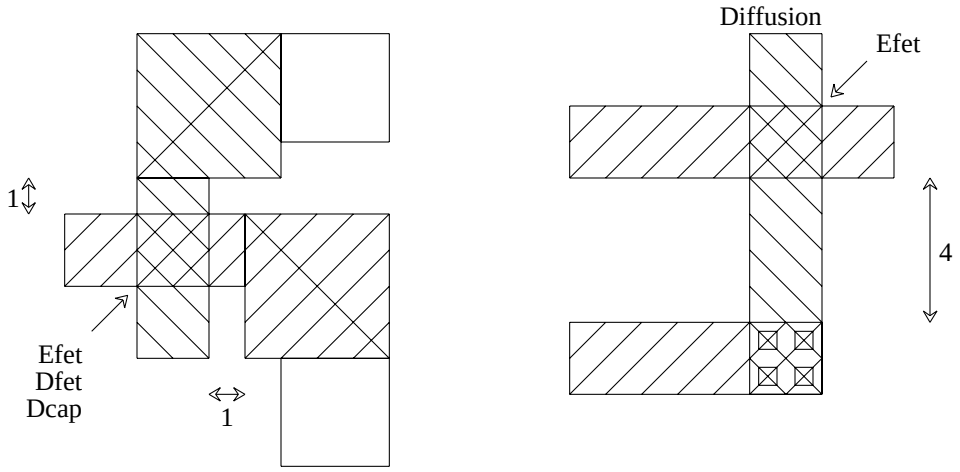
2.6. Buried Contacts



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

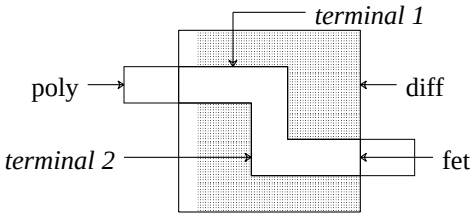
There is only one CIF output style available in the NMOS technology: **lambda=2**. The CIF layers in this style, and their meanings, are:

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

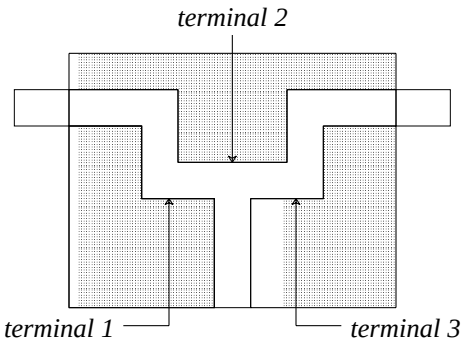
To see exactly where each CIF layer is generated for a particular design, use the **:cif see** command. There is also just one CIF input style. It is called **lambda=2** and can be used to read files written by Magic in the **lambda=2** style, or files written by Caesar using the standard NMOS technology 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.

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.

Other Reports In This Series

September, 1990

digital **Western Research Laboratory** 100 Hamilton Avenue Palo Alto, California 94301 USA

WRL Research Reports

‘‘Titan System Manual.’’
Michael J. K. Nielsen.
WRL Research Report 86/1, September 1986.

‘‘Global Register Allocation at Link Time.’’
David W. Wall.
WRL Research Report 86/3, October 1986.

‘‘Optimal Finned Heat Sinks.’’
William R. Hamburgen.
WRL Research Report 86/4, October 1986.

‘‘The Mahler Experience: Using an Intermediate Language as the Machine Description.’’
David W. Wall and Michael L. Powell.
WRL Research Report 87/1, August 1987.

‘‘The Packet Filter: An Efficient Mechanism for User-level Network Code.’’
Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta.
WRL Research Report 87/2, November 1987.

‘‘Fragmentation Considered Harmful.’’
Christopher A. Kent, Jeffrey C. Mogul.
WRL Research Report 87/3, December 1987.

‘‘Cache Coherence in Distributed Systems.’’
Christopher A. Kent.
WRL Research Report 87/4, December 1987.

‘‘Register Windows vs. Register Allocation.’’
David W. Wall.
WRL Research Report 87/5, December 1987.

‘‘Editing Graphical Objects Using Procedural Representations.’’
Paul J. Asente.
WRL Research Report 87/6, November 1987.

‘‘The USENET Cookbook: an Experiment in Electronic Publication.’’
Brian K. Reid.
WRL Research Report 87/7, December 1987.

‘‘MultiTitan: Four Architecture Papers.’’
Norman P. Jouppi, Jeremy Dion, David Boggs, Michael J. K. Nielsen.
WRL Research Report 87/8, April 1988.

‘‘Fast Printed Circuit Board Routing.’’
Jeremy Dion.
WRL Research Report 88/1, March 1988.

‘‘Compacting Garbage Collection with Ambiguous Roots.’’
Joel F. Bartlett.
WRL Research Report 88/2, February 1988.

‘‘The Experimental Literature of The Internet: An Annotated Bibliography.’’
Jeffrey C. Mogul.
WRL Research Report 88/3, August 1988.

‘‘Measured Capacity of an Ethernet: Myths and Reality.’’
David R. Boggs, Jeffrey C. Mogul, Christopher A. Kent.
WRL Research Report 88/4, September 1988.

‘‘Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description.’’
Deborah Estrin, Jeffrey C. Mogul, Gene Tsudik, Kamaljit Anand.
WRL Research Report 88/5, December 1988.

‘‘SCHEME->C A Portable Scheme-to-C Compiler.’’
Joel F. Bartlett.
WRL Research Report 89/1, January 1989.

‘‘Optimal Group Distribution in Carry-Skip Adders.’’
Silvio Turrini.
WRL Research Report 89/2, February 1989.

‘‘Precise Robotic Paste Dot Dispensing.’’
William R. Hamburgen.
WRL Research Report 89/3, February 1989.

''Simple and Flexible Datagram Access Controls for Unix-based Gateways.''
Jeffrey C. Mogul.
WRL Research Report 89/4, March 1989.

''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols.''
V. Srinivasan and Jeffrey C. Mogul.
WRL Research Report 89/5, May 1989.

''Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines.''
Norman P. Jouppi and David W. Wall.
WRL Research Report 89/7, July 1989.

''A Unified Vector/Scalar Floating-Point Architecture.''
Norman P. Jouppi, Jonathan Bertoni, and David W. Wall.
WRL Research Report 89/8, July 1989.

''Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU.''
Norman P. Jouppi.
WRL Research Report 89/9, July 1989.

''Integration and Packaging Plateaus of Processor Performance.''
Norman P. Jouppi.
WRL Research Report 89/10, July 1989.

''A 20-MIPS Sustained 32-bit CMOS Microprocessor with High Ratio of Sustained to Peak Performance.''
Norman P. Jouppi and Jeffrey Y. F. Tang.
WRL Research Report 89/11, July 1989.

''The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance.''
Norman P. Jouppi.
WRL Research Report 89/13, July 1989.

''Long Address Traces from RISC Machines: Generation and Analysis.''
Anita Borg, R.E.Kessler, Georgia Lazana, and David W. Wall.
WRL Research Report 89/14, September 1989.

''Link-Time Code Modification.''
David W. Wall.
WRL Research Report 89/17, September 1989.

''Noise Issues in the ECL Circuit Family.''
Jeffrey Y.F. Tang and J. Leon Yang.
WRL Research Report 90/1, January 1990.

''Efficient Generation of Test Patterns Using Boolean Satisfiablilty.''
Tracy Larrabee.
WRL Research Report 90/2, February 1990.

''Two Papers on Test Pattern Generation.''
Tracy Larrabee.
WRL Research Report 90/3, March 1990.

''Virtual Memory vs. The File System.''
Michael N. Nelson.
WRL Research Report 90/4, March 1990.

''Efficient Use of Workstations for Passive Monitoring of Local Area Networks.''
Jeffrey C. Mogul.
WRL Research Report 90/5, July 1990.

''A One-Dimensional Thermal Model for the VAX 9000 Multi Chip Units.''
John S. Fitch.
WRL Research Report 90/6, July 1990.

''1990 DECWRL/Livermore Magic Release.''
Robert N. Mayo, Michael H. Arnold, Walter S. Scott, Don Stark, Gordon T. Hamachi.
WRL Research Report 90/7, September 1990.

WRL Technical Notes

‘‘TCP/IP PrintServer: Print Server Protocol.’’
Brian K. Reid and Christopher A. Kent.
WRL Technical Note TN-4, September 1988.

‘‘TCP/IP PrintServer: Server Architecture and Implementation.’’
Christopher A. Kent.
WRL Technical Note TN-7, November 1988.

‘‘Smart Code, Stupid Memory: A Fast X Server for a Dumb Color Frame Buffer.’’
Joel McCormack.
WRL Technical Note TN-9, September 1989.

‘‘Why Aren’t Operating Systems Getting Faster As Fast As Hardware?’’
John Ousterhout.
WRL Technical Note TN-11, October 1989.

‘‘Mostly-Copying Garbage Collection Picks Up Generations and C++.’’
Joel F. Bartlett.
WRL Technical Note TN-12, October 1989.