# *MRouter* Reference Manual

Whiteley Research Incorporated
456 Flora Vista Avenue
Sunnyvale, CA 94086

Release 1.1.1
February 17, 2017

This page intentionally left blank.

# Contents

This page intentionally left blank.

# Chapter 1

# Introduction to *MRouter*

In this manual, text which is provided in `typewriter` font represents verbatim input to or output from the program. Text enclosed in square brackets ( [text] ) is optional in the given context, as in optional command arguments, whereas other text should be provided as indicated. Text which is *italicized* should be replaced with the necessary input, as described in the accompanying text.

## 1.1 *MRouter* Overview

The *MRouter* is an open-source maze router available from the free software archive on `wrcad.com`. It is derived from the *Qrouter* from `opencircuitdesign.com`.

The *MRouter* is a separate, stand-alone open-source project. Presently, users must download and build the *MRouter* tool. Instructions can be found in the README file in the distribution. Eventually, packages for the supported operating systems will be provided.

The *Qrouter* source code was converted from C to C++, and architected into two classes:

1. A database class. This caches technology and routing information. It can read and write technology and route information in standard LEF and DEF file formats. It makes use of the open-source Cadence LEF/DEF toolkit to provide complete and correct handling of these files.
2. The maze router itself. This is logically the same as the *Qrouter*, but the code has been refactored to make use of C++ encapsulation and other elements of the language. This will facilitate future development of new modes and features, and maintenance.

The two classes are exported as a plug-in library, intended to be used with *Xic*, or another application. *Xic* is a commercial graphical editor for integrated circuit design from Whiteley Research Inc. More information can be gound on the company's web site `http://wrcad.com`. *Xic* has a developing interface to the *MRouter* plug-in. providing full control of the routing operations, visualization, etc.

## 1.2 Theory of Operation

The *MRouter* is a maze router, which is terminology that describes a routing program that makes use of the Lee algorithm. On oversimplified description of the Lee algorithm is as follows.

1. The routing space is logically divided into an x-y grid of routing channels. We will assume that each layer has the same routing pitch for simplicity. From each routing grid location, one can logically move north, south, east, west, and up or down.
2. Obstructions and pins from placed gate instances are added to the grid by setting flags in the overlapping grid locations. These indicate locations where it is not possible to "travel".
3. Suppose you have a route with two connections: the source and the target. Both the source and target consist initially of a set of one or more points, each marked accordingly. One iteratively visits all of the non-obstructed grid points adjacent to a previously visited grid point, whth the initial points being the source. Iteration continues until a new point falls on a target location. We have then found a route through the maze.
4. Nets with more than two connections are handled analogously.
5. Once a complete route is found, we then "commit" the route by marking all of the grid points that the route covers, which are then treated like obstructions when finding routes on other nets.

Maze routers are most successful for small to medium designs, as they are compute and memory intensive. They do have the advantage that if a layout is routable, the Lee algorithm will find a route, and the route will be the lowest cost (perhaps shortest) of any alternatives.

*MRouter* (and *Qrouter*) use extensions to the basic Lee algorithm in order to reduce routing timer. First of all, routing is a two stage process. In stage 1, we route as much as can be routed, but typically some fraction of the nets will be painted into a corner and not be routable. We keep a "failed route" list of these routes. In stage 2, we use a different approach. For each route in the failed list, we will find a best route while ignoring collisions with existing routes. If this succeeds, we then find the routes that collide, rip them up, and add them to the end of the failed route list. We continue processing the failed route list until all nets are routed, or stage 2 can no longer make progress and stalls.

One important factor is the use of masking and multiple passes when finding routes. The mask identifies grid points that are candidates for searching for a route. We attempt to identify where a solution is likely to be found, and enable this area, plus some "slop" space. This minimizes the number of grid locations that need processing, speeding up the finding of nets, at least until congestion becomes heavy. As congestion becomes heavier, one has to increase the size of the mask open areas to find a solution. To find the last of the nets, one may work with no mask at all. This is resource consuming, but may be the only way to find nets that need to wander a long way off-course in order to finally make the connection.

When finding a route, we try an initial pass with a tight mask. If that fails, we perform subsequent passes, each time increasing the open area of the mask, until we have success or reach a limit on the number of passes.

## 1.3 Use with *Xic*

Once installed in the standard location, the *Xic* program from Whiteley Research Inc. will find and load the plug-in on start-up. A message "`Loading MRouter`" will be printed in the console along with the other startup messages.

When *Xic* starts, it will look for the *MRouter* plug-in in the following locations, in order.

1. The value of the `MROUTER_PATH` environment variable, if found. This is the full path to the shared library. If this variable exists, no other locations are checked.

2. The *MRouter* installation location, which defaults to `/usr/local/mrouter`. If installed in another location, the environment variable `MROUTER_HOME` can be set to the equivalent path. The library will be found in the `lib` subdirectory of the installation location.

There is a version compatibility test applied before the plug-in can be loaded. The *MRouter* version string consists of three integers separated by periods, as

*major*.*minor*.*release*

The *major* and *minor* numbers must match the version *Xic* was built for. If not, the plug-in is not loaded. The *release* number need not match, consequently there may be differences in behavior seen, but there should be no instability.

If the `XIC_PLUGIN_DBG` variable is set in the environment, messages are printed in the console tracing the plug-in search and load. This can be used to diagnosae a problem, as the process is otherwise silent.

Presently, the router is controlled exclusively with the **!mr** text command. The `MRouter` script function can be used to perform the operations from scripts.

This page intentionally left blank.

# Chapter 2

# Command Interface

The *MRouter* has a built-in set of command handlers that implement a basic command-line interface. Scripts can be produced by writing the commands in a file, one command per line. The scripts can be used for router setup, and initiation and control of routing jobs. Script files can read other script files to arbitrary depth.

The command processing is implemented hierarchically. At lowest level are the database commands, which are impolemented in the database class. Command processing for the router class will pass through unresolved operations to the database command processor. Finally, when using the *MRouter* plug-in, an application's command processing can pass through unresolved operations to the *MRouter* command processing. This allows the application to provide its own commands, or modified versions of the *MRouter* commands. For example, the *Xic* program adds commands to display the physical layout, and to move technology information to and from the *MRouter*.

The *Qrouter* has built-in TCL/TK support, and a comparable command interface. This support is not directly available in the *MRouter* presently. However, the command interface provided can be wrapped into TCL/TK, Python, or other script language functions, per the user's preference. Direct support for common languages such as TCL/TK and Python may be added in future releases.

## 2.1 The database commands

These are the commands handled by the database, and are available from an instance of the database class. They are also available from the router class, as a fallback.

### 2.1.1 Database commands: `reset`

**Syntax:** `reset`

This command takes no further arguments. It clears the database, and resets database parameter settings to default values.

### 2.1.2 Database commands: `set`

**Syntax:** `set` [*paramname* [*value*]]

The `set` command is used to set or query values for parameters used by the database and router. If no further arguments are given, a list of the parameters and their present values is printed. If a parameter name is given but no value, that parameter name and its current value (or its default) will be printed. Otherwise, the parameter whose name is given will be set to the given value.

The known parameters are as follows.

`debug` The *value* is a hex number whose bits are flags that turn on various debugging options. Presently undocumented.

`verbose`
This sets or displays the verbosity level for messages emitted by the database and router. If no *value* is given, the present verbosity level is printed. This is an integer in the range 0–4, where 4 is the most verbose. The *value* should otherwise be an integer in this range. The default verbosity is 0.

`global`
Up to six "global" nets can be specified. These are usually power or ground nets, to be treated specially by the router. Values are space-separated net names, which when specified are added to the internal list of global nets incrementally. That is, multiple calls can be made to define the global nets, or equivalently all of the global nets can be listed for a single call. The internal list can be cleared with `set clear global`. If no net names are given, the current global net name list is printed.

`vdd`
This is for *Qrouter* compatibility, for setting the name of the power net. This is actually equivalent to `global`.

`gnd`
This is for *Qrouter* compatibility, for setting the name of the ground net. This is actually equivalent to `global`.

layers
The *value* is an integer that represents the number o layers that the router will use. This can be set to a number less than or equal to the number of layers that have been defined (from LEF or technology input). For example, if given a value 4, only the first 4 routing layers defined will be used by the router. If no number is given, the number of layers that the router would use is printed.

`maxnets`
The *value* is the maximum number of nets allowed in a design. This can be set to limit the number of nets accepted for routing by the router. Jobs with too many nets will be rejected. If no number is given, the current maximum is printed.

`lefresol`
The *value* is the LEF resolution, as is normally provided in a LEF file. Acceptable values (from the LEF specification) are 100, 200, 400, 800, 1000, 2000, 4000, 8000, 10000, 20000. The internal coordinates use this many points per micron. The resolution is 100 if not otherwise given. If not given, the current value is printed. Otherwise, the resolution may be set to the value given, however once the LEF resolution is set, whether by LEF file or this keyword, it can not be changed except by resetting the database.

`mfggrid`
The *value* is the manufacturing grid usually obtained from a LEF file. It is provided as a real

number in microns. The value must be exactly representable in the LEF resolution, and will quantize coordinate values. The *value* can be 0, in which case no manufacturing grid will be used, which is the default. If no *value* is given, the present manufacturing grid, in microns, is printed.

`definresol`
The *value* is the resolution as normally read from a DEF file. Acceptable values are as listed for `lefresol`, and furthermore the value must exactly divide the `lefresol`. The resolution is 100 if not otherwise given. If not given, the current value is printed. Otherwise, the resolution may be set to the value given, however once the DEF input resolution is set, whether by DEF file or this keyword, it can not be changed except by resetting the database.

`defoutresol`
The *value* is the resolution used when writing DEF files with `write def`, which defaults to the `definresol`. Setting to 0 clears the overriding. If no *value*is given, the present DEF output resolution override is printed.

`netorder`
The *value* can be set to one of the following integer values. It controls the order in which nets are routed.

]item0
Use the default ordering, where nets with the most taps are routed first.

1
Use alternate ordering, where the nets with the largest minimum box containing taps are routed first.

2
Retain the ordering as nets were defined.

If no *value* is given, the present ordering enumerator is printed.

`passes`
The *value* is an integer that is the maximum number of attempted passes of the route search algorithm, where the routing constraints (maximum cost and route area mask) are relaxed on each pass. The default number of passes is 10. With no *value*, the maximum number of passes is printed. Otherwise, a positive integer is expected for the *value*.

`increments`
The *value* is a list of positive integers, each giving the incremental halo width in the routing mask for the corresponding routing pass. The mask is used to limit the area where a route can be implemented. On each pass, this area is expanded by the corresponding `increments` value. The final increments value is used for any subsequent passes. The default is effectively the single integer "1", so that the masking area is bloated by one channel on each pass. If no integers are given, the internal list of increments is printed.

`via_stack`
The *value* sets the maximum number of vias that may be stacked directly on top of each other at a single point. A *value* of "`none`" or "`0`" or "`1`" will disallow stacked vias. A *value* of "`all`" will allow all vias to be stacked (which is the default). Otherwise, the *value* should be an integer less than or equal to the number of via layers.

`via_pattern`
If vias are not square, then they are placed in a checkerboard pattern, with every other via rotated 90 degrees. If inverted, the rotation is swapped relative to the grid positions used in the non-inverted case. If the *value* is 0 or a word starting with "`n`", the pattern is not inverted. If the *value* is nonzero or a word starting with "`i`", the pattern is inverted.

### 2.1.3 Database commands: `setcost`

**Syntax:** `setcost` [*paramname* [*value*]]

This is similar to the `set` command, but is exclusively for the cost parameters used to optimize routing. Advanced users and experimenters may appreciate the ability to access these parameters. The cost parameters are the following. For most of these, only the first character is checked. If the first character is '`c`', the second character is also checked.

`segcost`
Add description. The default is 1.

`viacost`
Add description. The default is 5.

`jogcost`
Add description. The default is 10.

`xvercost` or `crossovercost`
Add description. The default is 4.

`blockcost`
Add description. The default is 25.

`offsetcost`
Add description. The default is 50.

`conflictcost`
Add description. The default is 50.

### 2.1.4 Database commands: `unset`

**Syntax:** `unset` *keyword* [*keyword* ...]

The arguments are the keywords that are accepted by the **set** or **setcost** commands. At least one must be given, but any number can appear in the argument list. For each keyword, the default values are set, undoing any **set** operation.

### 2.1.5 Database commands: `ignore`

**Syntax:** `ignore` [*netname* ...] [`-u` [`all`]] [*netname* ...]

The database contains a list of net names that will be ignored by the router. The list is maintained by this command.

With no options, The entire list or names of ignored nets is printed. Otherwise, the arguments consist of net names or the special token `-u`. Net names given before or without `-u` will be added to the ignored net list (if not already present). Net names given after `-u` will be removed from the ignore net list, if found.

A special case is when the special keyword "`all`" follows `-u`. In this case, the entire list is cleared. This is not undoable so proceed with caution.

### 2.1.6 Database commands: `critical`

**Syntax:** `critical` [*netname ...*] [`-u` [`all`]] [*netname ...*]

The database contains a list of net names that will be considered critical by the router. Such nets will be afforded higher priority by the router. The list is maintained by this command. With no options, The entire list or names of critical nets is printed.

With no options, The entire list or names of ignored nets is printed. Otherwise, the arguments consist of net names or the special token `-u`. Net names given before or without `-u` will be added to the critical net list (if not already present). Net names given after `-u` will be removed from the critical net list, if found.

A special case is when the special keyword "`all`" follows `-u`. In this case, the entire list is cleared. This is not undoable so proceed with caution.

### 2.1.7 Database commands: `obstruction`

**Syntax:** `obstruction` [`-u`] [*layer*] [`all`] [*L B R T*]

The database contains a list of rectangular areas for each routing layer that are given to the router as obstructions. These are user-defined areas where routes are not allowed to intersect. The obstruction list is maintained by this command.

If no arguments appear, the current list of user-defined obstructions is printed. If the `-u` option is given, the command will potentially remove obstructions from the list, otherwise obstructions will be added (if an identical obstruction is not already present). The *layer* can be either the name of a routing layer, or its 1-based index number.

The special keyword `all` can appear only when `-u` is given. When it appears, if a *layer* was given, all obstructions on that layer will be removed from the list. If no *layer* is given, then the entire obstruction list is cleared.

Otherwise, if a *layer* is given, but the coordinates are omitted, a list of the obstructions on the given layer is printed. If the box coordinates are included (given in microns), the indicated rectangle is added to the list if `-u` was not given, or removed from the list (if it exists) if `-u` was given.

### 2.1.8 Database commands: `layer`

**Syntax:** `layer` [*layer* [*option*] [*args*]]

This command sets or queries some basic parameters of routing layers. If no arguments are given, a listing of the routing layers with their present parameter values is printed.

The first argument is a layer name or 1-based routing layer index. If only this is given, the parameters for the given layer are printed.

The *option* that may follow indicates the parameter. It can be one of the following.

`-n`
: This indicates the layer name. If a word follows, the layer will be given this name. If no word follows, the present layer name will be printed.

`-l`

This indicates the user layer number. This may indicate the GDSII layer number. It is a user parameter and not used by the router. It must be a non-negative integer if given. If a number follows, it will be assigned, otherwise the present value is printed.

`-t`
This indicates the user type number. This may indicate the GDSII datatype number. It is a user parameter and not used by the router. It must be a non-negative integer if given. If a number follows, it will be assigned, otherwise the present value is printed.

`-w`
This indicates the default wire width used for routing, in microns. It must be a positive number. If a number follows, it will be assigned, otherwise the current value is printed.

`-p`
This indicates the pitch of the routing, meaning the wire width plus minimum spacing between wires. This must be a positive value larger than the wire width. If a number follows, it will be assigned, otherwise the current value is printed.

`-d`
This indicates the preferred route direction, which can be horizontal of vertical. If a word follows, the first character of which is '`h`' or '`v`', the direction is set. Otherwise, the current direction is printed.

### 2.1.9 Database commands: `newlayer`

**Syntax:** `!mr newlayer` *name*

This creates a new layer with the given name and appends it to the layers list. The **layer** command can then be used to fill in the routing parameters.

### 2.1.10 Database commands: `boundary`

**Syntax:** `boundary` [*L B R T*]

This sets or prints the bounding box of the routed area in microns. This is ordinarily set from the cell and technology data.

### 2.1.11 Database commands: `read`

**Syntax:** `read` *source* [*filename*]

This command will read a file into the database. The *source* keyord must be one of the following.

`config`
This will read a configuration file in the format used by the *Qrouter* from which *MRouter* was derived. If no *filename* is given, the name "`route.config`" is assumed. If given, the *filename* should be a path to a Qrouter-style config file. This functionality is deprecated and may be removed at some point.

`script`
A script is a text file containing commands, as described in this section, one per line. The *filename*

should be a path to such a file. The file will be read, and the commands executed in order, unless a fatal error occurs, which terminates execution. Calls to read scripts can recurse to any depth.

`lef`
The *filename* should be a path to a file in the Cadence LEF format (Layout Exchange Format). This is an industry-standard format used to provide routing information about the standard cells to be used by the router. This includes technology information about the routing layers (e.g., routing direction, pitch, width), and/or definitions of the standard cells including size information and terminal locations. It may also provide definitions for the standard vias defined for the process.

`def`
The *filename* should be a path to a file in the Cadence DEF format (Design Exchange Format). This is an industry-standard format used to pass netlist information. This will include placement locations and orientations of the cell instances used in the design, plus a list of nets which define the circuit. The DEF file provides the nets in abstract form, the router implements these nets in physical form.

### 2.1.12 Database commands: `write`

**Syntax:** `write` *target* [*filename*]

The `write` command will create a file. The *target* can be one of the following.

`lef`
The *filename* is the name of a LEF file to write. The file will contain the technology and standard-cell information found in the database.

`def`
The *filename* is the name of a DEF file to write. The file will contain the components, pins, and nets as saved in the database.

### 2.1.13 Database commands: `append`

**Syntax:** `append` *deffile1 deffile2*

This will copy the DEF file *deffile1*, while adding the physical routing information, to *deffile2*. This is pretty much the same as the *Qrouter* output method. The *deffile1* must be the original input DEF file, or contain the same instance placements and abstract routes. When the router is run, the physical routes are written back to the database, where they can be used to update the original source as described.

## 2.2 The router commands

When called through the router object, the list of available commands is a superset of the database commands listed in the previous section. The `read script` command is overloaded to handle scripts containing router commands.

### 2.2.1 Router commands: `reset`

**Syntax:** `reset` [`all`]

This overrides the database command of the same name. If the "`all`" argument is given, the router and database are both reset to a pristine state. If `all` is not given, only the router is reset, database information is retained. In this context, `true`, `yes`, and `1` (one) are synonyms for `all`, case insensitive, and only the leading character is actually read.

### 2.2.2 Router commands: `stage1`

**Syntax:** `stage1` [*options*]

Execute stage1 routing. This works through the entire netlist, routing as much as possible but not doing any rip-up and re-route. Nets that fail to route are put in the failed nets list. The number of failed routes is printed.

The following options are understood.

`-d`
: Draw the area being searched in real-time. This slows down the algorithm and is intended only for diagnostic use. **Not implemented.**

`-s`
: Single-step stage one, i.e., do one route per call, incrementally.

`-m`[ ]*value*
: The *value* follows the option character with or without intervening white space. It can be an integer 0 or larger, and will set the search mask size. It can also be set to one of the following keywords, where only the first character is significant.

  `n`[`one`]
  : Don't limit the search area.

  `a`[`uto`]
  : Select the mask automatically.

  `b`[`box`]
  : Use the net bounding box as a mask.

`-f`
: Force a terminal to be routable.

*netname*
: Route the named net only.

### 2.2.3 Router commands: `stage2`

**Syntax:** `stage2` [*options*]

Execute stage2 routing. This stage works through the failed nets list, routing with collisions, and then ripping up the colliding nets and appending them to the failed nets list.

The number of failed nets remaining is printed on exit.

The following options are understood.

`-d`
: Draw the area being searched in real-time. This slows down the algorithm and is intended only for diagnostic use. **Not implemented.**

`-s`
Single-step stage two, i.e., do one route per call, incrementally.

`-m[ ]`*value*
The *value* follows the option character with or without intervening white space. It can be an integer 0 or larger, and will set the search mask size. It can also be set to one of the following keywords, where only the first character is significant.

`n[one]`
Don't limit the search area.

`a[uto]`
Select the mask automatically.

`b[box]`
Use the net bounding box as a mask.

`-f`
Force a terminal to be routable.

`-l` *N*
Fail route if the solution collides with more than *N* nets.

`-t` *N*
Keep trying *N* additional times.

*netname*
Route the named net only.

### 2.2.4 Router commands: `stage3`

**Syntax:** `stage3` [*options*]

Execute stage3 routing. This works through the entire netlist, ripping up each route in turn and re-routing it. The number of failed nets remaining is printed on exit.

The following options are understood.

`-d`
Draw the area being searched in real-time. This slows down the algorithm and is intended only for diagnostic use. **Not implemented.**

`-s`
Single-step stage three, i.e., do one route per call, incrementally.

`-m[ ]`*value*
The *value* follows the option character with or without intervening white space. It can be an integer 0 or larger, and will set the search mask size. It can also be set to one of the following keywords, where only the first character is significant.

`n[one]`
Don't limit the search area.

`a[uto]`
Select the mask automatically.

`b`[`box`]
Use the net bounding box as a mask.

`-f`
Force a terminal to be routable.

`-l` *N*
Fail route if the solution collides with more than *N* nets.

`-t` *N*
Keep trying *N* additional times.

*netname*
Route the named net only.

### 2.2.5 Router commands: `ripup`

**Syntax:** `ripup` [`-a` | *netmname* ...]

Rip up (remove all physical routing information from) a net or nets, or all nets in the design. If `-a` is specified, all nets will be ripped up. Otherwise, the arguments are names of nets to rip up.

### 2.2.6 Router commands: `failed`

**Syntax:** `failed` [*option*]

The default operation, i.e., with no argument given, is to print the number of nets currently in the failed nets list. Otherwise, one of the following options may be given.

`-a`
Move all nets to the failed net list, ordered by the standard metric.

`-u`
Move all nets to the failed nets list, as originally ordered.

`-p`
Print a list of failed net names.

### 2.2.7 Router commands: `congest`

**Syntax:** `congest` [`-n` *N*] [*filename*]

This command will compute and print a congestion value for each cell instance. The values are print in sorted order, highest congestion to lowest. By default, all instances are listed, if the `-n` option is given, only the highest *N* instances are listed (*N* being a positive integer). If a *filename* is given, output goes to that file, otherwise output goes to the standard output. This command is intended to identify highly congested parts of a layout, which may benefit from additional fill in a subsequent routing run.

# Chapter 3

# Additional Stuff

Add text here, lots of it.

This page intentionally left blank.

# Index

This page intentionally left blank.