Circuit Design Aids (CDA) on Plan 9

A. G. Hume

M. Kahrs

T. J. Killian

ABSTRACT

CDA is a system for the design and prototyping of digital systems. At the front end it provides hierarchical schematic entry, programmable logic device design and board layout; at the back end it produces data for various manufacturing technologies, in particular wire-wrap and multiwire.

1. Introduction.

CDA is a design system, i.e., a collection of programs and data formats dating back almost 15 years. It has progressed with the accompanying changes in display, computing and device technology. To the CDA user, A hardware design has a logical part and a physical part. The logical part consists of circuit schematics, generally supplemented by boolean equations together with finite state machines and programs in ROMs. The physical part includes board layout and wire routing.

CDA has its own terminology; a circuit contains *chips* each identified by a *name* (which is arbitrary, and of mnemonic value to the designer) and a *type* (which is generic, e.g., `74LS74`). Schematics can be hierarchical; what appears syntactically as a chip can, in fact, be an instance of a parameterless *macro*, (i.e., another drawing) if the file *type*`.w` exists. Real chips have *pins*, each identified by a *pin name* and *pin number*, and a *package type*. Pin names and their mapping onto pin numbers are a property of the chip type; the mapping from pin numbers to physical coordinates is a property of the package type.

Pins are connected by *nets*, which have unique *net names* (assigned by the drawing to net conversion program if omitted by the user). It is an error for a pin to be connected to more than one net. Nets such as `VCC` and `GND` generally need different routing algorithms from ordinary nets; these are called *special-signal nets* in cases where the distinction is important.

A *board* is a physical mounting for packages. It is mostly characterized by its *pin holes* (available for package insertion) and *special-signal pins* (connected to special-signal nets). An *I/O connector*, where signals enter or leave the board, is simply a special case of a chip.

The manual pages for the CDA commands and file formats are in section 10 of the manual. Coventionally, the commands are kept in `/bin/cda`, and so, for example, to run the *gnet* program, you would actually type something like

```
cda/gnet < timing.g > timing.w
```

2. Methodology.

These are the conventional steps in a design. Many are necessary simply to maintain consistency between ‘‘source’’ and ‘‘object’’ files. We will collect all of this into a `mkfile` in a later section.

(1) The interactive program *graw* is used to construct schematics (kept in files whose names end with `.g`.) The net list of a circuit diagram (its `.w` file) are derived from the `.g` file by running *gnet*.

(2) Any editor may be used to create files in `lde` format for logic that is to be implemented with Programmable Array Logic (PAL)’s. These filenames end with `.lde`. Pin information resides in a

corresponding `.p` file, generated by *part* which is a member of the *part* family of programs.

(3) A `.pins` file, that matches pin names with numbers for each chip type, must be constructed. Most pin information comes from standard libraries, but the user must generally supply some of it, usually for I/O connectors (`io.pins`) or non-standard chips (`my.pins`). *Mkpins* reads `.w` files, `.p` files, and pin libraries to produce the `.pins` file. The principal advantage of using *mkpins* is to reduce the size of the pins file and thereby speeding up the time spent in *cdmglob.*

(4) *Cdmglob -f -v* reads the `.w` and `.pins` files to produce a `.wx` file, in which all macros are expanded, and nets are described in terms of pin numbers. The `−v` flag tells *cdmglob* to include the name of the expanded pins in the output. This will be used in the final stages by *annotate* to create a `.a` file containing just the pin numbers (in *graw* format). This file, when "catted" with a `.g` file will label all the pins with pin numbers.

(5) At this point one may do static circuit checks with *smoke*. The errors will be rather voluminous until all pins are declared correctly on the `.tt` lines. Some errors are impossible to eradicate, particularly those with a mix of analog and digital components.

(6) Most files discussed so far have to do with the logical part of the design, and, except for `.lde` files, are in CDL (Circuit Design Language). The remainder of the physical design files are in FIZZ format. So, at this point, one uses *fizz cvt* to turn the `.wx` file from *cdmglob* into a FIZZ `.fx` file.

(7) As with the `.pins` file, one creates a `.pkg` file with geometric descriptions of each package type.

(8) A geometric description of the board (`.brd` or `.board` file) in FIZZ format is made (or stolen from `/sys/lib/cda/boards`).

(9) Chip positioning information (`.pos` file) is generated. This is usually done interactively with *place*.

(10) At this point, the design should be checked with *check.* This will find any errors that might result from unplaced chips or overlapping packages and so forth.

(11) The wrap list (`.wr` file) is now made, and one can physically wrap the board, typically by using a semiautomatic machine.

(12) To make changes, one generates a new `.wr` file; *rework* then compares the new and old wrap files and generates separate lists for unwrapping and rewrapping.

3. Graphics input

The graphics editor *graw* is used to create and modify drawings, a.k.a. schematics. A drawing consists of *chips, macros,* and *signals* connected by *pins.* Each chip has a *name* and a *type.* Pins can have either a *name* or a *number.*

3.1. Using graw

The editor, *graw* can be given a list of files ending in *.g.* or an empty list. When *cda* starts, the cursor changes to a ×. Button 1 performs two tasks: a single click locates the cursor; when dragged with the button held down, the mouse leaves behind a line. Button 2 presents a list of useful options: `onesies`→ can be used to select either `box` which then can be used to sweep out the rectangle of a box using button 1, or `macros` which can also sweep out a box using button 1.

`inst`→ selects a master to be instantiated and attached to the cursor until any button is pressed. *graw* doesn't have any masters when loaded initially. The standard library of gates can be read by using the `read` command. `sweep` uses a rectangle input with button 1 to grab a set of objects and drag them until any button is pressed.

`slash` differs from `sweep` only in that rectilinear lines are first cut by the input rectangle.

`cut` undraws and moves the object(s) last drawn or moved to the cut/paste buffer. `paste` attaches a copy of the cut/paste buffer to the cursor until any button is pressed.

`snarf` is a `cut` without the undraw.

`scroll` attaches the entire drawing to the cursor until any button is pressed.

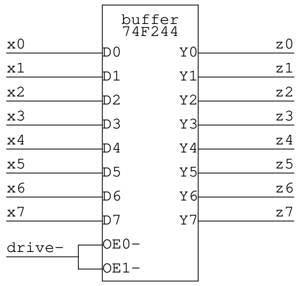
The button 3 menu entries are `edit, read, write, exit,` and `new,` followed by the list of filenames currently being edited.

`edit` prompts for a file name and reads in the file for editing. Backspace and control-W may be used to edit the name; a null file name aborts the operation. `read` prompts for the name of a master file, reads it in, and plants a reference to it in the current file. The names of the masters in the file are added to those in the `inst`→ menu for the current file, overwriting older definitions if necessary. `write` prompts for a file name (starting with the current file name). The non-null result becomes the new file name and the file is written. `exit` terminates the program. You must type a ‘y’ to really exit. `new` creates a new, unnamed drawing for editing.

Selecting a file name selects the current file.

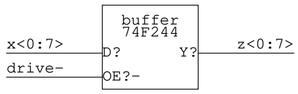
4. Signal Bundles and Macros.

Consider this buffer between two 8-bit busses:



It illustrates several *graw* conventions. The *chip* is indicated by a box; its *name* is `buffer`; its *type* is `74F244`. These are simply unattached text strings that appear stacked inside the box. *Pin names* (e.g., `D0`) are strings that appear on the inside edge of the box. *Nets* are lines that end on a pin. *Net names* are strings that are placed on nets. A trailing − conventionally indicates an active-low signal.

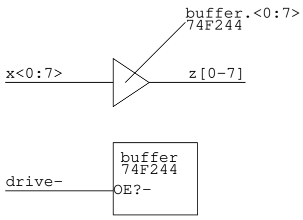
Even this trivial example involves repeated patterns. A much more succinct equivalent is:



The *generator* `x<0:7>` expands into the ordered list `x0`, `x1`,... `x7`. The *pattern* `D?` matches two-character pin names that begin with `D`. (The space of possible names comes from the `.pins` file entry for the chip type.) The names that match the pattern are *sorted alphabetically* and put into correspondence with the nets.

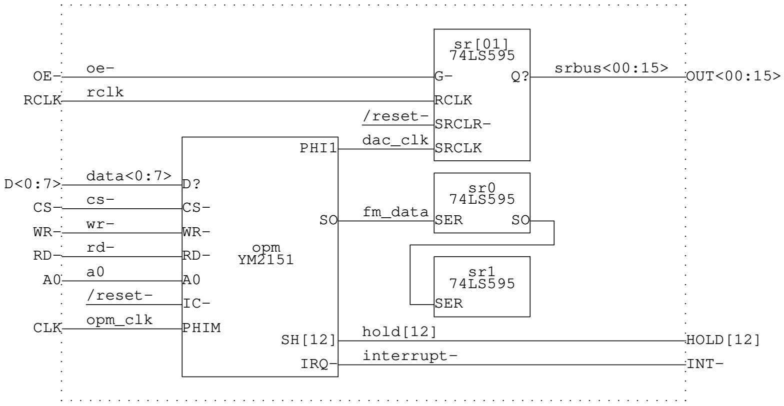
If connected sets of nets and pins do not have the same cardinality, the smaller set is reused until the larger is exhausted. Thus in the example, the `drive-` net gets connected to both `OE0-` and `OE1-`, as desired.

Another example of the same buffer is shown below:



Note that since the buffer symbol is too small to hold the name of the chip, the name and type are "connected" to the instance via a wire. The wire is *not* considered a net by *gnet*. Also notice that the range <0:7> after the name of the chip is appended to the pin names. In fact, this buffer has invisible pin names of `D` and `Y`. (input and output respectively). Thus, after appending the range, *gnet* will generate `D<0:7>` and `Y<0:7>`. Since the output enable can’t fit, the output enables are put in another, smaller box.

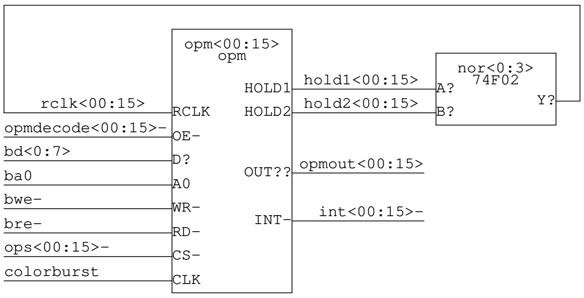
Frequently one has a group of chips that will be used or replicated as a unit. In such a case it makes sense to define a *macro* that may be instantiated as required. A macro lives in its own file. Here is an example, `opm.j`:



The dotted box (produced by sweeping out a `macro` box in *graw*) identifies the circuit as a macro. Strings outside of this box (conventionally in upper case) are ‘‘pin names’’ visible to the outside world. Most names inside the box will be made local to each instantiation. Net names beginning with / are ‘‘globals,’’ i.e., they represent the same signal throughout the design. `/VCC` and `/GND` are the most common global signals. These signals are expanded by *cdmglob* to be ‘‘<instance>//VCC’’ and ‘‘<instance>//GND’’ respectively. A *sed* script can be used to rid the net list of the file name prefixes:

```
////s/         .*/// /
s/       //      /
```

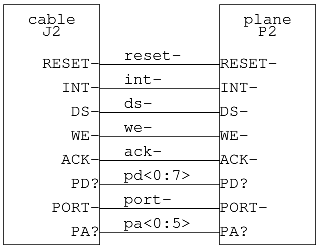
Now we use generators to make several instances of `opm.j`:



Sixteen copies of the `opm` circuit are made. The pattern `D?` is at a lower, i.e., ‘‘faster running,’’ level than `opm<00:15>`, with the effect that all the `D0`’s are connected to `bd0`, all the `D1`’s are connected to `bd1`, etc; similarly, all the `A0`’s are connected to `ba0`. On the other hand, all the `CS–`’s are separate: `opm00/CS–` (the instance of `CS–` in `opm00`) is connected to `ops00–`, `opm01/CS–` is connected to `ops01–`, etc. The manual entry for *cdmglob* should be consulted for all the details.

5. A Toy Example.

In this section we present a complete example. The design consists of two I/O connectors that route signals from a ribbon cable to a backplane. Here is the schematic, followed by the `.w` file:




```
.c      cable   J2
        reset-  ,RESET- % 6 64 56
        int-    ,INT-   % 6 64 72
        ds-     ,DS-    % 6 64 88
        we-     ,WE-    % 6 64 104
        ack-    ,ACK-   % 6 64 120
        pd<0:7> ,PD?    % 6 64 136
        port-   ,PORT-  % 6 64 152
        pa<0:5> ,PA?    % 6 64 168
.c      plane   P2
        pa<0:5> ,PA?    % 4 144 168
        reset-  ,RESET- % 4 144 56
        int-    ,INT-   % 4 144 72
        ds-     ,DS-    % 4 144 88
        we-     ,WE-    % 4 144 104
        ack-    ,ACK-   % 4 144 120
        port-   ,PORT-  % 4 144 152
        pd<0:7> ,PD?    % 4 144 136
```

The comments (introduced by `%`) are coordinates from the `.g` file that can be used later to annotate the drawing with pin numbers. Otherwise the `.w` file is mostly just a compendium of the text strings in the `.g` file. In order to proceed further, we need a `.pins` file:

```
.t J2 BERG40 % plugs into ribbon cable
%       .   10    .   20    .   30    .   40
.tt 2959292g5g4g4g4g4g4g4g4g4g2g2g2g2g292929
.tp VCC[1-4] 2 4 38 40
.tp V12 6
.tp V12- 36
.tp GND0[1-9] 8 10 12 14 16 18 20 22 24
.tp GND1[0-4] 26 28 30 32 34
.tp RESET- 1
.tp INT- 3
.tp DS- 5
.tp WE- 7
.tp ACK- 9
.tp PD[0-7] 11 13 15 17 19 21 23 25
.tp PA[0-5] 27 29 31 33 35 37
.tp PORT- 39

.t P2 DIN96RX % plugs into backplane
%       .   10    .   20    .   30     .   40    .   50    .   60     .   70    .   80    .   9
.tt vgigig5g4g4gv44gigigigigigigiggv VGnnnnnnnnnGVnnnnnnnnGnnnnnnnnGV vg5gig4g4g4gv4igigigigigi
.tp RESET- 3
.tp INT- 67
.tp DS- 5
.tp WE- 69
.tp ACK- 7
.tp PD[0-7] 71 9 73 11 75 14 78 15
.tp PA[0-5] 79 17 81 19 83 21
.tp PORT- 85
.tp PC[0-7] 23 87 25 89 27 91 29 93
.tp GND<00:14> 2 4 6 8 10 12 16 18 20 22 24 26 28 30 31
.tp GND<15:18> 34 44 54 63
.tp GND<19:33> 66 68 70 72 74 76 80 82 84 86 88 90 92 94 95
```

It should be fairly obvious what is going on here. Note the appearance of the *package type* following the type name on the `.t` line.

For some parts, constructing a `.tt` line will be an onerous task (typical examples are parts with PGA pinouts). For these parts, the use of *pga* is recommended. It accepts a list of pin names and pin types (one per line) and produces a suitable part definition. The manual page has all the details.

Now the `.wx` file can be made with `cdmglob -v -f` to get:

```
.t J2 BERG40
.tt 2959292g5g4g4g4g4g4g4g4g4g2g2g2g2g292929
.t P2 DIN96RX
.tt vgigig5g4g4gv44gigigigigigigiggv VGnnnnnnnnGVnnnnnnnnGnnnnnnnnGV vg5gig4g4g4gv4igigigigigi
.f      toy.w
.c      plane   P2
        reset-  3       RESET-
        ds-     5       DS-
        ack-    7       ACK-
        pd1     9       PD1
        pd3     11      PD3
        pd5     14      PD5
        pd7     15      PD7
        pa1     17      PA1
        pa3     19      PA3
        pa5     21      PA5
        int-    67      INT-
        we-     69      WE-
        pd0     71      PD0
        pd2     73      PD2
        pd4     75      PD4
        pd6     78      PD6
        pa0     79      PA0
        pa2     81      PA2
        pa4     83      PA4
        port-   85      PORT-
.c      cable   J2
        reset-  1       RESET-
        int-    3       INT-
        ds-     5       DS-
        we-     7       WE-
        ack-    9       ACK-
        pd0     11      PD0
        pd1     13      PD1
        pd2     15      PD2
        pd3     17      PD3
        pd4     19      PD4
        pd5     21      PD5
        pd6     23      PD6
        pd7     25      PD7
        pa0     27      PA0
        pa1     29      PA1
        pa2     31      PA2
        pa3     33      PA3
        pa4     35      PA4
        pa5     37      PA5
        port-   39      PORT-
```

This is basically a listing, for each chip, of net name−pin number pairs. Package types are copied through from the `.pins` file for use by the physical design tools, and the expanded pin names are left as an aid to humans and *annotate*.

At this stage, physical layout can begin. The first step is the conversion of the `.wx` file into a FIZZ `.fx` file via *cvt*. The result is shown below:

```
Type{
   name J2
   pkg BERG40
   tt 2959292g5g4g4g4g4g4g4g4g4g2g2g2g2g292929
}
```

```
Type{
   name P2
   pkg DIN96RX
   tt vgigig5g4g4gv44gigigigigigigiggvVGnnnnnnnnGVnnnnnnnnGnnnnnnnnGVvg5gig4g4g4gv4igigigigigigigiggv
Chip{  name plane  type P2  }
Chip{  name cable  type J2  }
Net pd0 2{  cable 11 PD0  plane 71 PD0  }
Net pd1 2{  cable 13 PD1  plane 9 PD1  }
Net pd2 2{  cable 15 PD2  plane 73 PD2  }
Net pd3 2{  cable 17 PD3  plane 11 PD3  }
Net pd4 2{  cable 19 PD4  plane 75 PD4  }
Net pd5 2{  cable 21 PD5  plane 14 PD5  }
Net pd6 2{  cable 23 PD6  plane 78 PD6  }
Net pd7 2{  cable 25 PD7  plane 15 PD7  }
Net ds- 2{  cable 5 DS-  plane 5 DS-  }
Net int- 2{  cable 3 INT-  plane 67 INT-  }
Net port- 2{  cable 39 PORT-  plane 85 PORT-  }
Net reset- 2{  cable 1 RESET-  plane 3 RESET-  }
Net we- 2{  cable 7 WE-  plane 69 WE-  }
Net ack- 2{  cable 9 ACK-  plane 7 ACK-  }
Net pa0 2{  cable 27 PA0  plane 79 PA0  }
Net pa1 2{  cable 29 PA1  plane 17 PA1  }
Net pa2 2{  cable 31 PA2  plane 81 PA2  }
Net pa3 2{  cable 33 PA3  plane 19 PA3  }
Net pa4 2{  cable 35 PA4  plane 83 PA4  }
Net pa5 2{  cable 37 PA5  plane 21 PA5  }
```

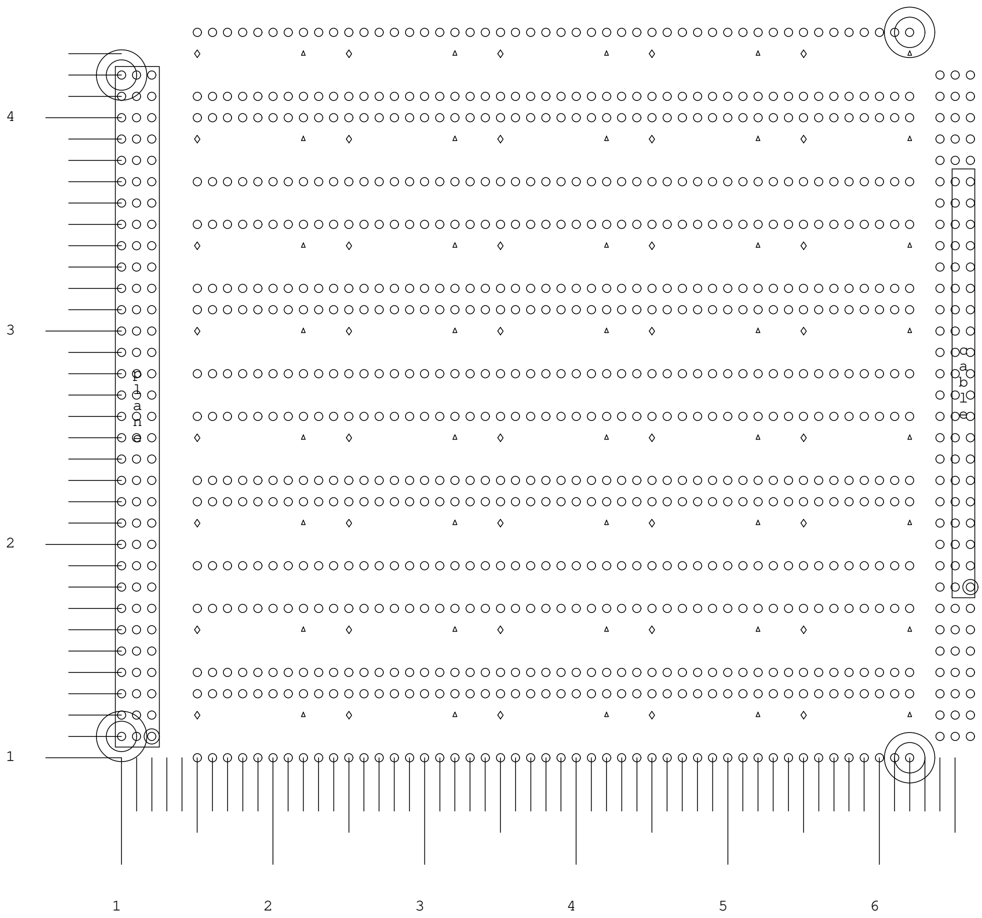
Next, a board file is selected. Since this is a simple example, a simple board like a Schroff 3U board can be used. Here’s the definition of the Schroff board:

```
Board{
        name schroff_board
        align 1000/1100 6200/1000 6200/4400 1000/4200
}
Pinholes {
        1500/1000 4800 400 100/300 A
        1500/1400 4800 400 100/300 A
        1500/1900 4800 400 100/300 A
        1500/2300 4800 400 100/300 A
        1500/2800 4800 400 100/300 A
        1500/3200 4800 400 100/300 A
        1500/3700 4800 400 100/300 A
        1500/4100 4800 400 100/300 A
}
Pinholes {
        1000/1100 300 3200 100/100 A
        6400/1100 300 3200 100/100 A
}
Vsig 0{
        name GND
        pins 40{
                1 2200/1200 - 5 6200/1200 V
                6 2200/1600 - 10 6200/1600 V
                11 2200/2100 - 15 6200/2100 V
                16 2200/2500 - 20 6200/2500 V
                21 2200/3000 - 25 6200/3000 V
                26 2200/3400 - 30 6200/3400 V
                31 2200/3900 - 35 6200/3900 V
                36 2200/4300 - 40 6200/4300 V
        }
}
Vsig 1{
        name VCC
        pins 40{
                1 1500/1200 - 5 5500/1200 V
                6 1500/1600 - 10 5500/1600 V
                11 1500/2100 - 15 5500/2100 V
                16 1500/2500 - 20 5500/2500 V
                21 1500/3000 - 25 5500/3000 V
                26 1500/3400 - 30 5500/3400 V
                31 1500/3900 - 35 5500/3900 V
                36 1500/4300 - 40 5500/4300 V
        }
}
```

To fully understand this definition, a careful reading of the FIZZ format manual page is advised. Briefly, the align is used with the semi-automatic wirewrap machine. The pinholes locate pins − the letter `A` denotes a drill type (ignored here but still required). The final two `Vsig` declarations specify the special signals. Given a board and the `.fx` file, *place* can be used to place these two connectors `BERG40` and `DIN96RX`. The result is a `.pos` file like this:

```
Positions{
        plane 1200/1100 0 0
        cable 6600/1800 1 16
}
```

Check should be run on these files (`.board`, `.fx`, `.pos`) and any errors should be resolved before proceeding. *Plot* generates a picture of the board:



Note how the alignment marks are shown. Pin 1 of each package is illustrated by two concentric circles around the pins. Now a wrap file (with extension `.wr`) suitable for driving the semi-automatic wire-wrap machine can be created with *wrap -cv*. The use of Multiwire technology needs some hand holding − consult an expert.

6. Debugging and rework

Debugging a circuit is beyond the scope of this paper, but CDA does provide support for altering a wirewrap board.

The scenario is that you have a board that corresponds to a wraplist `v1.wr`. After making the necessary changes to your schematics and or logic equations, a new wraplist `v2.wr` is generated. You then run the command

```
cda/rework v1.wr v2.wr
```

which generates three files. The first file, `UN.wr`, contains a wraplist of the wires to take off (remove). The second file, `RE.wr`, contains a wraplist of the wires to add. The third file, `NEW.wr`, is a wraplist that is electrically equivalent to `v2.wr` but represents the wirewraps on the board (that is, if you wrapped a board with NEW.wr you would get an identical board). After doing the unwrapping and rewrapping, you should treat NEW.wr as your actual v2.wr.

7. Analog design

In spite of the fact that CDA was created for the design of digital circuits, it can also be utilized for analog circuits as well. A separate library of analog shapes located in /lib/graw/analog.g. These shapes follow the following naming convention: the shape name begins with an orientation, a underscore, and then the name of the shape. This is because graw doesn't have rotation or reflection operations (yet). The characters denote:

character	orientation
h	horizontal
v	vertical
l	left
r	right
t	top
b	bottom

For example, an electrolytic capacitor with the positive terminal is called "t_ecap". An NPN transistor with the emitter on the bottom left would be "bl_npn". Analog design is facilitated by the famous -k option to gnet. This option eliminates the need to draw lines to each component. The only problem is figuring out the best location for the part name. Only experience will help you.