Pcb

an interactive
printed circuit board
layout system for `X11`

Thomas Nau

Copying

History

`Pcb` is a handy tool to develop printed circuit board layouts. It was first written for an Atari ST in 1990 and ported to `UNIX` and `X11` in 1994. I never had the intention to create another professional layout system but to write a tool which supports people like me who do some home-developing of hardware. For this reason no auto-router and/or auto-placement code has been added.

The second release 1.2 includes menus for the first time. This hopefully makes `PCB` easier to use and for that a more important tool.

Release 1.3 introduced an undo-feature for the most destructive commands, a more straight forward action handling and scalable fonts.

Special thanks goes to:

Bernhard Daeubler (Bernhard.Daeubler@physik.uni-ulm.de)
Harald Daeubler (Harald.Daeubler@physik.uni-ulm.de)
Roland Merk (merk@faw.uni-ulm.de)
Erland Unruh (Erland.Unruh@malmo.trab.se)

who did most of the alpha and beta testing, helped to port `PCB` to several operating systems and platforms and corrected several typos in the manuals.

1 Introduction

Each layout consists of several, mostly independent, objects. This chapter gives an overview about existing types and their relation to each other. For a complete description refer to Chapter 2 [Getting Started], page 5. You may also want to look at Chapter 6 [File Formats], page 26. All distances and sizes are in mil (0.001 inch). The origin is in the upper left corner of the screen, x grows to the right, y to the bottom. In contrary the y axis grows from bottom to top if you use `PostScript` printouts. The sections in this chapter are sorted by the order of appearance of the objects within a layout file.

1.1 Symbols

The top object is the layout itself. It uses a set of symbols called a font which resides at the first logical level. Each symbol is uniquely identified by a 7 bit ASCII code. All other layout objects share the same set of symbols. Undefined symbols are drawn as filled rectangles.

Every font file is preprocessed by a user-defined command at load-time. For details see '`fontCommand`', Section 5.1 [Resources], page 16.

1.2 Vias

Vias are identical to the pins used by elements except that they can be removed individually. It's possible to assign a name to a via even if it doesn't make to much sense. Their only purpose is to connect layers. They should not be used for adding circuits to the layout even if it seems to be easier than to create a new element.

1.3 Elements

Elements represent all electrical circuits of a layout. They also reside on the first logical level like symbols which means that they are shared by all under-laying levels. Elements are loaded from ASCII coded files very similar to the layout file itself. They are composed of lines and arcs to define the package, two labels which define the canonical name of the element as well as the name used in the layout, a mark to make positioning easier and pins which also have labels. Right now it's not possible to create SMD layouts with this kind of elements because the pins are always connected to all layers. All parts of an element are treated as one unit. It's not possible to delete a single pin or break the element into pieces. This object is also supported by two special layers, *pkg. outline* and *pkg. pins*, which are used to toggle displaying of the package and pins.

Every element file is preprocessed by a user-defined command at load-time. For details see '`elementCommand`', Section 5.1 [Resources], page 16. Using `m4` for example allows you to create libraries for package definitions which are shared by all circuits. The files `circuits/*` shipped with this release use this mechanism. `Pcb` is able to create a list of all connections from one (or all) elements to the others or a list of unconnected pins.

1.4 Layers

Every layout consist of several layers which might be used independently or grouped together for switching them on and off. For details see '`fileCommand`', Section 5.1 [Resources], page 16. Each layer is drawn in a user defined color and is identified by its name. They are a kind of containers for line, polygon and text objects. The connections to other layers are established by element pins or by vias. There is no relation between a specific layer and the board itself. The user has to decide which layer is the solder side and which one will be the component side. You can also use different layers for one board side to separate the power lines from signal lines. The release 1.3 and later handles all layers in a group identical when scanning for connections.

1.5 Lines

Lines are used for three different purposes. First to connect pins and/or vias, second to draw a package of an element and last to draw symbols. Automatic clipping to 45 degree lines is available. Remember, the y axis grows downwards on a `X11` display and therefor the y coordinates have a different meaning.

1.6 Polygons

It's quiet useful sometimes to fill large areas. The only way to do so is to use polygons. `Pcb` can handle all types of polygons but using a number of smaller ones improves performance while scanning connections. Editing in form of deleting or inserting points is supported.

1.7 Text

Text objects should be used to label a layout or to put some additional information on the board but not to identify elements (circuits) which do have their one labels (they appear in connection lists too). The drawing direction for text objects are 0, 90, 180 and 270 degrees. A flag determines if it should be mirrored when it is drawn.

TEXT OBJECTS CREATE COPPER LINES BUT THEY ARE NOT SCANNED FOR CONNECTIONS.

2 Getting Started

The goal of this chapter is to give you enough information to learn how `Pcb` works and how to develop your layouts to make best use of its features. All event translations refer to the shipped application default resource file. There is probably no need to change them except if your window manager uses some of the button events itself. `Pcb` will write some information to *stderr* or to the logging window during its lifetime.

Get yourself a printout of this chapter and *User Commands*, if you haven't done so, and follow the examples. An example layout can be found in `example/68HC11`.

Start `Pcb` without any additional options from the distribution directory to have access to the font file. An error message

can't find default font-symbol-file 'FONTFILENAME'

indicates a wrong font searchpath or filename in the application resource file. Check if your `m4` support searchpaths. If not, get GNU `m4`. For other messages see Section A.2 [problems], page 32.

2.1 The Application Window

The main window consists of three areas:

2.1.1 The Statusline and Input-field

The statusline is located at the bottom edge of the main window. During normal operation the status information is always visible. If a selected menu operation requires an additional button click, the statusline will be replaced by a information message.

It displays, from left to right, the cursor position, the current values of grid with *r* for relative and *a* for absolute mode and an optional second *a* for lines with any direction. The grid information is followed by scaling, linewidth, viasize and drilling hole in mil, the currently used buffer and the name of the layout. An asterisk which appears at the left edge indicates that the layout has been modified.

The inputline pops up whenever user input is required. Two keys are bound to the input field: *None<Key>Escape* aborts the input, *None<Key>Return* finishes it successfully.

2.1.2 The Control Panel

The control panel, located at the left side of window, is used to display and change active layers as well as the current drawing layer. If a layers hasn't been named the label *(unknown)* will be used instead.

The upper control box is used to switch layers on and off. Click *None<Btn1Down>* to one or more of them. Each click toggles the setting. The layout is redrawn when the pointer reenters the drawing area to prevent from excessive redrawing when the visibility of several layers is changed. The currently active layer can't be switched off. If you have already installed some layer groups, clicking to these fields will toggle the visibility of all layers of the same group.

The lower control box, named *current*, is used to change the current drawing layer. Try *None<Btn1Down>* on some of the fields. Each of them is labeled with the layers name and drawn with its color. The new drawing layer is always switched on. Try and change the current layer's name to *ABC* selecting *set name of layer* from the *File* menu. Changing the active layer is also available as *!Mod1<Key>1..MAX_LAYER*.

2.1.3 The Mode Selectors

The mode selector icons reside right at the bottom of the control panel. They are used to select operation mode of `Pcb`. All of them should be self explaining. *None<Btn1Down>* causes the appropriate action to be taken. The drawing modes are also available by:

None<Key>Escape	`reset mode`
None<Key>F1	`via-mode`
None<Key>F2	`line-mode`
None<Key>F3	`pastebuffer-mode`
None<Key>F4	`rectangle-mode`
None<Key>F5	`text-mode`
None<Key>F6	`polygon-mode`

The cursor shape changes depending on the selected mode.

2.1.4 Drawing Area

The drawing area is made of a viewport widget which also includes two scrollbars. Moving the pointer into it changes the cursor shape depending on the current operation mode. A crosshair follows the X pointer with respect to the grid setting. Move around and watch the cursor position displayed at the bottom. Now select a new grid from the *Display* menu. The new value is updated in the statusline. Now move again and watch the difference. A different way to change the grid is *!Shift<Key>g* to decrement or *<Key>g* to increment it. The setting is saved together with the data. For normal, homemade layouts a value of 10 or 50 is a good setting. The cursor can also be moved with the cursor keys or, for larger distances, by pressing the *shift* modifier together with a cursor key.

2.1.5 Menu

All menus are located at top of the drawing area. Only a few of their functions are available from the keyboard too. Some of the entries like *center* require a certain cursor position. In this case a info line will popup at the bottom saying something like

> press button at position or key to abort

Any mouse button will do the job whereas any key beside the cursor keys will cancel the operation. For details see Section 5.2 [Actions], page 20.

'`About`' There is no menu hiding behind this button but a small information box.

'`File`' This menu offers a selection of loading, saving and printing data, saving connection information to a file or quitting the application. You may also change the layouts or the current layers name. Selecting *PostScript* pops a a printer control dialog which is, hopefully, self explaining. This box contains a panner widget (only `X11R5` and later) which simplifies adjusting the offsets. With earlier releases the printout will always appear in the upper left corner with respect to the media margins.

'`Display`' The display menu supports the most needed functions that are related to screen output. The entries are used to change the grid to some popular values, the zoom factor, the displayed element name as well as to center or redraw the output. You may also switch displaying of the grid on or off and select between absolute grid (origin at (0,0)) or relative grid (origin at position where grid has been changed).

‘`Sizes`’ This menu changes the initial size of new vias, drilling holes and lines as well as the current maximum size of the layout.

‘`Objects`’ Displaying the pinout of an element and changing it’s names as well as editing a text object is offered by this menu. The last two selections require an additional pointer button click at the objects position.

‘`Selection`’
This menu covers most of the operations that work with selected objects. You may either (un)select all visible objects of a layout or only the ones which have been found by the last connection scan. Toggling the selection of a single object is available by pressing *None*<*Btn3Down*>. *!BTNMOD*<*Btn3Down*>, moving and *!BTNMOD*<*Btn3Up*> select all visible objects inside the rectangle. Pressing the modifier key *Shift* too unselects all objects in the area. The other entries change the sizes of visible and selected objects.

‘`Buffer`’ This menu handles pastebuffer related actions. You may select one out of 5 buffers to use, rotate or clear its contents and paste it to the layout. Be aware that only visible objects are pasted to the layout.

‘`Connections`’
The entries available through this menu button allow the user to find connections from pins or vias and to manipulate these. The connection lists can be saved by selecting entries from the *File* menu.

‘`Undo`’ This menu is a frontend for managing undo’s off destructive operations like remove, copy, move and the changing of names. The number of operations is unlimited (depending on memory). The list is cleared if new layout data is loaded or by an entry in this menu.

2.2 Log Window

This optional window is used to display all kind of messages. These include the ones written to *stderr* by external commands. The main advantage is that its contents can be searched and that they are available till the program exits. Disabling this feature by setting the resource *useLogWindow* to *false* will generate popup windows to display messages. The *stderr* of external commands will appear at `Pcb`s *stderr* which normally is the parent shell. I suggest that you iconify the window after startup. If *raiseLogWindow* is set *true*, the window will deiconify and raise itself when new messages are to be displayed.

2.3 Drawing and Removing Basic Objects

There are some ways of creating new objects. First, you draw them yourself or second, by copying an existing object and third by loading an element from a file. Creating new objects is normally related to a special mode depending on the object type. The notation of key and button events is the same as described in the X11 Intrinsics manual.

The operation mode can be selected by one of the mode selectors in the bottom left corner or by one of the function keys listed earlier in this chapter. *None<Btn1Down>* and *None<Key>space* send a notify request to the application which responds by creating or changing the appropriate object or at least takes the first step to do so. Selecting a mode forces the mode selector to be redrawn with a different line thickness.

Removing objects is possible by *None<Key>Backspace* which removes the object at the cursor location. If more than one object is located at the same position, the smallest type will be selected. If two or more are of the same type, the newest one will be chosen. You may also change to *remove-mode* and click *None<Btn1Down>* at the location of the objects which are to be removed.

Mirroring and rotating work similar to that. Change the mode and press *None<Btn1Down>* at the objects location. Remember, not all object types support mirroring and rotating.

Removing objects, changing their size or moving them does only apply to the ones that are visible when the command is entered.

There are several keystrokes and button events that refer to an *object* without identifying its type. Here's a list of them:

None<Key>space and *None<Btn1Down>* create an object depending on the current mode.

None<Key>BackSpace removes the visible object at the cursor location. The order is: pin, via, line, text, polygon and element. Only one object is removed. If two or more of the same type match, the newest one is removed.

Use *None<Key>s* and *!Shift<Key>s* to change the size (width) of lines, text objects, pins and vias.

None<Key>n changes the name of pins, vias and elements or the string of a text object.

None<Key>u recovers from an unlimited number of destructive operations like removing, moving or copying objects.

For a complete list of keystrokes and button events checkout Section 5.3 [Translations], page 25.

2.3.1 Lines

To draw new lines you have to be in *line-mode*. Get there by either selecting it from the *Modes* menu or by pressing *None<Key>F2*. The statusline shows the new setting. Each successive press of a *create* button creates a new line of a polygon. The adjustment to 45 degree lines is done automatically if it is selected from the *Display* menu. Press *None<Key>Escape* to leave line-mode.

None<Key>l and *!Shift<Key>l* and the menu entries of *Sizes* change the initial value of new lines that is displayed in the statusline.

2.3.2 Polygons and Rectangles

A polygon is drawn by defining all of its segments like drawing lines. If the first point matches a new one and if the number of points is greater than 2 the the point is assumed the final one. Since relocating the first point may be a hard job you can use *!BTNMOD<Btn1Down>* to close the polygon. An error can occur if clipping to 45 degree lines has been selected and the final segment does not match this condition. The *rectangle-mode* is just a easy way to generate rectangles. All of them are handled like polygons internally. I suggest that you create small convex polygons to avoid an impact on the scanning routines and since editing of polygons isn't supported yet. *Polygon-mode* is also selected by *None<Key>F6* whereas *rectangle-mode* uses *None<Key>F4*. Pressing a *None<Btn1Down>* at two locations creates it by defining two of its corners. *None<Key>Insert* inserts a new point into an existing polygon. The new point is located at the crosshair location and is placed on the segment with the lowest distance. *!Shift Ctrl<Key>Insert* removes a point of a polygon if the number of corners is greater then 3. Pressing *None<Key>p* while entering a new polygon brings you back to the previous corner wheras *!Shift<Key>p* closes the polygon if possible. The last command may fail if clipping to 45 degree lines is enables. Inserting or removing a point don't force clipping.

2.3.3 Text

Pressing *None<Key>F5* or clicking into the selector button changes to *text-mode*. Each successive notify event (*None<Btn1Down>*) pops up the input line at the bottom and queries for a string. Now enter it and press *None<Key>Return* to confirm or *None<Key>Escape* to abort. The text object is created with its upper left corner at the current pointer location. The cursor location is the fixpoint for rotations.

Now switch to *rotate-mode* or *mirror-mode* and press *None<Btn1Down>* at the textobjects location. Mirroring textobjects will be used for strings on the solder side of the layout.

Use *None<Key>n* to change the string.

2.3.4 Vias

The initial size of new vias can be changed by *None<Key>v* and *!Shift<Key>v* or by selecting the appropriate entry from the *Sizes* menu. *!Mod1<Key>v* and *!Mod1 Shift<Key>v* do the same for the drilling hole of the via. The statusline is updated with the new values. Creating a via is similar to the other objects. Switch to *via-mode* by using the selector button of *None<Key>F1* then press *None<Key>Space* of *None<Btn1Down>* to create one. *None<Key>n* changes the name of a via.

2.3.5 Elements

Some of the functions related to elements only work if both the package layer and the pin layer are switched on.

First of all, you have to load data into the paste buffer. There are two ways to do so:

load the data from a file
copy data from an already existing element

We don't have any elements on the screen yet so we use number one.

`:le` pops up a fileselector box which is self-explaining. Select *mc68030_pga* from the *circuits* directory. The data is loaded and the mode is switched to *pastebuffer-mode*. If the circuit couldn't be found ask your SysAdmin to check the search paths. Each notify event does now create one of these biests. Just reset the mode by unselecting the mode button or by *None<Key>Escape*. The crosshair is located at the *mark* position as defined by the data file. Rotating the buffer contents is done by selecting the *rotate* entry of the *Buffer* menu. The contents of the buffer are valid until new data is loaded into it either by a cut-and-paste operation or by loading a new data file. There are MAX_BUFFER buffers available. Switching between them is done by selecting a menu entry or by *!Shift<Key>1..MAX_BUFFER*.

The release includes all data files for the circuits that have been used by the demo layout. If you have problems with the color of the crosshair change the resource setting to a different one.

Now load a second data file called `mc68882_pga` by entering *`:le mc68882_pga`*. Create the circuit as explained a few lines up. You now have two different unnamed circuits, a CPU and a FPU. Unnamed means the the layouts name of the element hasn't been set yet. Selecting *canonical* from the *Display* menu displays the canonical names of the two circuits which are MC68030 and MC68882. Each of the two names of an element can be changed by *None<Key>n* at the elements location and editing the old name in the bottom input line. Naming pins and vias is similar to elements.

The mentioned second way to create a new element is to copy an existing one. Please refer to Section 2.4 [Moving and Copying], page 10.

To display the pinout of a circuit move to it and push *None<Key>d* or select *show pinout* from the *Objects* menu. A new window pops up and displays the complete pinout of the element. Dismiss it by clicking to the appropriate button. You may also want to change a pins current size by pressing *None<Key>s* to increment or *!Shift<Key>s* to decrement it.

For information on element connections refer to Section 2.7 [Connection Lists], page 12.

2.3.6 Pastebuffer

The linestack and element-buffer of former releases have been replaced by MAX_BUFFER multi-purpose buffers which are selected by *!Shift<Key>1..MAX_BUFFER*. The statusline shows the currently selected one. You can load data from a file into them or copy layout data. Cut-and-paste works too. If you have followed the instructions earlier in this chapter you should now have several objects on the screen. Move the crosshair to one of them and press *None<Btn3Down>* to toggle it's selection flag. The object is redrawn in a different color. You may also want to try *!BTNMOD<Btn3Down>*, move the pointer while holding the button down and release it on a different location. This selects all objects inside the rectangle. Using *!Shift Mod1<Btn3Down>* unselects all objects. Now change to *pastebuffer-mode* and select the entry of the *Buffer* menu which you want to try. Copying objects to the buffer is available as *!Ctrl<Key>x* while cutting them uses *!Ctrl Shift<Key>x*. Both clear the buffer before new data is added. Element data or PCB data can be merged into an exosting layout by loading the datafiles into the pastebuffer. Both operations are available from the *File* menu or as user commands.

2.4 Moving and Copying

All objects beside pins can be moved, even element-names, by *None<Btn2Down>*, moving the pointer while holding the button down and releasing it at the new location of the object. If you use *!BTNMOD<Btn2Down>* instead the object is copied. This does not work for element-names. You have to use the cut-and-paste paradigm for copying and moving groups of objects. Please refer to Section 2.3.6 [Pastebuffer], page 10.

2.5 Loading and Saving

After your first experience with `Pcb` you will probably like to save your work. *`:s name`* passes the data to an external program which is responsible for saving it. For details see *saveCommand* in Section 5.1 [Resources], page 16. Saving is also available from the *File* menu, either with passing a filename or using the original one. `Pcb` reuses the last filename if you do not pass a new one to the save routine.

To load an existing layout either select *load layout data* from the *File* menu or use *`:l filename`*. A file selectbox pops up if you don't specify a filename. Merging existing layouts into the new one is supported either by the *File* menu or by *`:m filename`*.

`Pcb` make a backup of the current layout depending on the resource *backup*. The file is named `BACKUP_NAME`. During critical sections of the program or when data would be lost it is saves as `EMERGENCY_NAME`. *%i* is replaced by the process ID.

2.6 Printing

`Pcb` is only able to create a PostScript file. I recommend the use of `GhostScript` if you don't have a PostScript printer. Select either *visible layers* or *package, pins and vias* from the *File* menu. Both of them pop up a control panel with the following options:

'`scaling`' It's quiet useful to enlarge your printout for checking the layout. Use the scrollbar to adjust the scaling factor to your needs.

'`rotate`' Rotate layout 90 degrees counter-clockwise before printing (default).

'`mirror`' Mirror layout before printing. Use this option for the solder side depending on your production line.

'`color`' Pass color information to the PostScript file. All colors will be converted to black if this option is inactive.

'`media`' Select the size of the output media from this menu. The user defines size can be set by the resource *media* either from one of the well known paper sizes or by a `X11` geometry specification. This entry is only available if you use `X11R5` or later. For earlier releases the user defined size or, if not available, *A4* is used.

'`offset`' Adjust the offsets of the printout by using the panner at the right side of the dialog box. This entry is only available if you use `X11R5` or later. A zero offset is used for earlier releases.

'`commandline`'
Use this line to enter a command (starts with `|`) or a filename. The default is set by the resource *printCommand*.

The created file includes some labels which are guaranteed to stay unchanged

'`PCBMIN`' identifies the lowest x and y coordinates in mil.

'`PCBMAX`' identifies the highest x and y coordinates in mil.

'`PCBOFFSET`'
is set to the x and y offset in mil.

'`PCBSCALE`'
is a floating point value which identifies the scaling factor.

`PCBSTARTDATA`
`PCBENDDATA`
: all layout data is included between these two marks. You can use them with a `awk` script to produce several printouts on one piece of paper by duplicating the code and putting some `translate` commands in front. Be aware that the normal `PostScript` units are 1/72 inch.

2.7 Connection Lists

After completing parts of your layout you may want to check if all drawn connections match the ones you have in mind. To demonstrate how to use of the appropriate commands execute the following stuff:

```
create at least two elements and name them
create some connections between their pins
optionally add some vias and connections to them
```

Now select *find* from the *Connections* menu, move the cursor to a pin or via and press any mouse button. `Pcb` will look for all other pins and/or vias connected to the one you are at and display the objects in a different color. Now try some of the reset options available from the same menu.

There is also a way to scan all connections of one element. Select *a single element* from the menu and press any button at the elements location. All connections of this element will be saved to the specified file. Either the layout name of the element or its canonical name is used to identify pins depending on the one that is displayed on the screen (can be changed by *Display* menu).

An automatic scan of all elements is initiated by choosing *all elements*. It behaves like scanning a singe element except that the resource *resetAfterElement* is used to determine if connections should be reset before a new element is scanned. Doing so will produce very long lists because the power lines are rescanned for every element. The default is set to *false* for this reason.

Scanning for unconnected pins is selected as *unused pins* from the same menu.

2.8 Selection

Some commands mentioned earlier in this chapter can also operate on all selected and visible objects. Now go back to the layout and toggle the selection flag of a single one by *None<Btn3Down>*. Try *!BTNMOD<Btn3Down>*, move the pointer while holding the button down and release it on a different location. This selects all objects inside the rectangle. Using *!Shift Mod1<Btn3Down>* unselects all objects.

The entries of the *Selection* menu are self-explaining.

3 User Commands

The entering of user-commands is initiated by the action routine *Command()* and finished by either *None<Key>Return* or *None<Key>Escape* to confirm or to abort. These two key-bindings cannot be changed from the resource file. The triggering event, normally a key press, is ignored. The input area will replace the bottom statusline. It pops up when *Command()* is called. The arguments of the user-commands are passed to the external commands without modification. See also the resource *saveInTMP*.

There are simple *usage* dialogs for each command and one for the complete set of commands.

'`l [filename]`'
: Loads a new datafile and, if confirmed, overwrites any existing unsaved data. The filename and the searchpath (*filePath*) are passed to the command defined by *fileCommand*. If no filename is specified a file select box will popup.

'`le [filename]`'
: Loads an element description into the paste buffer. The filename and the searchpath (*elementPath*) are passed to the command defined by *elementCommand*. If no filename is specified a file select box will popup.

'`m [filename]`'
: Loads an layout file into the paste buffer. The filename and the searchpath (*filePath*) are passed to the command defined by *fileCommand*. If no filename is specified a file select box will popup.

'`q[!]`' Quits the program without saving any data (after confirmation).

'`s [filename]`'
: Passes data and filename to the command which was set by the resource *saveCommand*. If no filename entered either the last one is used again or, if it is not available, a file select box pops up.

4 Command-Line Options

There are several resources which can be set or reset beside the standard toolkit command-line options. For a complete list refer to Section 5.1 [Resources], page 16.

The synopsis is:

```
pcb [-option ...] [-toolkit_option ...] [layout-file]
```

or

```
pcb -specialoption
```

4.1 Options

'`-alldirections/+alldirections`'
: Disables or enables line clipping to 45 degree angles. Overwrites the resource *allDirectionLines*.

'`-backup value`'
: Time between two backups in seconds. Passing zero disables the backup feature. Overwrites the resource *backupInterval*.

'`-c value`'
: Number of characters per output line. The resource *charactersPerLine* is overwritten.

'`-fontfile filename`'
: The default set of symbols (font) for a new layout is read from this file. All directories as defined by the resource *fontPath* are scanned for the file. The scan is only performed if the filename doesn't contain a directory component. The *fontFile* resource is changed.

'`-lelement command-line`'
: Sets the command to be executed when an element is loaded from a file to the paste buffer. The command may contain %f and %p to pass the requested filename and the searchpath to the command. It has to write the data to its standard output. The related resource is *elementCommand*.

'`-lfile command-line`'
: Sets the command to be executed when a new layout is loaded from a file. The command may contain %f and %p to pass the requested filename and the searchpath to the command. It has to write the data to its standard output. The related resource is *fileCommand*.

'`-lfont command-line`'
: Sets the command to be executed when a font is loaded from a file. The command may contain %f and %p to pass the requested filename and the searchpath to the command. It has to write the data to its standard output. The related resource is *fontCommand*.

'`-lg layergroups`'
: This option overwrites the resource *layerGroups*. See its description for more information. The value is used for new layouts only.

'`-loggeometry geometry`'
: Determines the geometry of the log window.

'`-pnl value`'
: Restricts the displayed named of a pin in the pinout window to the passed value. See also the resource *pinoutNameLength*.

'`-pz value`'
: Sets the zoom factor for the pinout window according to the formula: scale = 1:(2 power value). The related resource is *pinoutZoom*.

'`-reset/+reset`'
: If enabled, all connections are reset after each element that is scanned. The feature is only used while scanning connections to all elements. See also *resetAfterElement*.

'`-ring/+ring`'
: Overrides the resource *ringBellWhenFinished*. If enabled, the bell on the keyboard is rang when connection searching has finished.

'`-s/+s`' Enables/Disables the saving of the previous commandline. Overrides the *saveLastCommand* resource.

'`-save/+save`'
: See the resource description of *saveInTMP* for details.

'`-sfile command-line`'
: Sets the command to be executed when an layout file is saved. The command may contain %f which is replaced by the filename. The command must read it's data from the standard input. The resource *saveCommand* is overwritten.

'`size <width>x<height>`'
: Overrides the resource *size* which determines the maximum size of a layout.

'`-v value`' Sets the volume of the X speaker. The value is passed to `XBell()` and has to be in the range -100..100.

4.2 Special Options

There are some special options available in addition to normal command-line options. Each of them can only be specified without any other options. The available special options are:

'`-copyright`'
: Prints out the copyright notice and terminates.

'`-version`'
: Prints out the version ID and terminates.

'`-help`' Prints out the usage message and terminates.

5 X11 Interface

This chapter gives an overview about the additional `X11` resources which are defined by `Pcb` as well as the defined action routines. A widget tree is also included.

5.1 Non-Standard X11 Application Resources

Beside the toolkit resources, Pcb defines the following additional resources:

‘`absoluteGrid (boolean)`’
: Selects if either the grid is relative to the position where it has changed last or absolute, the default, to the origin (0,0).

‘`backupInterval (int)`’
: `Pcb` has an automatic backup feature which saves the current data every n seconds. The default is *300* seconds. A value of 0 disables the feature. The backup file is named `BACKUP_NAME`. *%i* is replaced by the process ID. See also the command-line option *-backup*.

‘`charactersPerLine (int)`’
: `Pcb` uses this value to determine the page width when creating lists. N, the number of characters per line, defaults to *80*. See also the command-line option *-c*.

‘`connectedColor (color)`’
: All pins, vias, lines and rectangles which are selected during a connection search are drawn with this color. The default value is determined by *XtDefaultForeground*.

‘`crosshairColor (color)`’
: This color is used to draw the crosshair cursor. The color is a result of a *XOR* operation with the contents of the drawing area. The result also depends on the default colormap of the used `X11` server because only the colormap index is used in the boolean operation and `Pcb` doesn't create it's own colormap. The default setting is *XtDefaultForeground*.

‘`elementColor (color)`’
: The elements package part is drawn in this color which also defaults to *XtDefaultForeground*.

‘`elementCommand (string)`’
: `Pcb` uses a user defined command to read element files. This resources is used to set the command which is executed by the users default shell. Two escape sequences are defined to pass the selected filename (%f) and the current search path (%p). The command has to write the element data to its standard output. The default value is

  M4PATH="%p";export M4PATH;echo 'include(%f)' | m4

  Using the GNU version of `m4` is highly recommended. See also the command-line option *-lelement*.

'`elementPath (string)`'
: A colon separated list of directories or commands (starts with '`|`'). The path is passed to the program specified in *elementCommand* together with the selected elementname. A specified command will be executed in order to create entries for the fileselect box. It has to write its results to *stdout* one entry per line. See also the user-command *le[!]*.

'`fileCommand (string)`'
: The command is executed by the users default shell whenever existing layout files are loaded. Data is read from the commands standard output. Two escape sequences can be specified to pass the selected filename (%f) and the current search path (%p). The default value is

  cat %f

  See also the command-line option *-lfile*.

'`filePath (string)`'
: A colon separated list of directories or commands (starts with '`|`'). The path is passed to the program specified in *fileCommand* together with the selected filename. A specified command will be executed in order to create entries for the fileselect box. It has to write its results to *stdout* one entry per line. See also the user-command *l[!]*.

'`fontCommand (string)`'
: Loading new symbol sets is also handled by an external command. You may again pass the selected filename and the current search path by passing %f and %p in the command string. Data is read from the commands standard output. This command defaults to

  cat %f

  See also the command-line option *-lfont*.

'`fontFile (string)`'
: The default font for new layouts is read from this file which is searched in the directories as defined by the resource *fontPath*. Searching is only performed if the filename does not contain a directory component. The default filename is *FONTFILENAME*. See also the command-line option *-fontfile*.

'`fontPath (string)`'
: This resource, a colon separated list of directories, defines the searchpath for font files. See also the resource *fontFile*.

'`grid (int)`'
: This resources defines the initial value of one cursor step. It defaults to *100 mil* and any changes are saved together with the layout data.

'`layerColor1..MAX_LAYER (color)`'
: These resources define the drawing colors of the different layers. All values are preset to *XtDefaultForeground*.

'`layerGroups (string)`'
: The argument to this resource is a colon separated list of comma separated layernumbers (1..MAX_LAYER). All layers within one group are switched

on/off together. The default setting is *1:2:3:...:MAX_LAYER* which means that all layers are handled separatly. Grouping layers one to three looks like *1,2,3:4:...:MAX_LAYER* See also the command-line option *-lg*.

'`lineThickness (dimension)`'
: The value, range [MIN_LINESIZE,MAX_LINESIZE], defines the initial thickness of new lines. The value is preset to *10 mil*.

'`media (<predefined> | <width>x<height>)`'
: The default (user defined) media of the PostScript device. Predefined values are *a3*, *a4*, *a5*, *letter*, *tabloit*, *ledger*, *legal*, and *executive*. The second way is to specify the medias width and height in mil. The resource defaults to *DEFAULT_MEDIASIZE* size.

'`mediaMarginLeft (int)`'
'`mediaMarginBottom (int)`'
: The resources determine the device dependent offset of a printout. Both default to *500*.

'`offLimitColor (color)`'
: The area outside the current maximum settings for width and height is drawn with this color. The default value is determined by *XtDefaultBackground*.

'`pinColor (color)`'
: This resource defines the drawing color of pins. The value is preset to *XtDefaultForeground*.

'`pinoutFont (string)`'
: This font is used to display pin names in the pinout window. Its value is preset to *XtdefaultFont*.

'`pinoutNameLength (int)`'
: This resource limits the number of characters which are displayed for pin names in the pinout window. By default the string length is limited to *8* characters per name. See also the command-line option *-pnl*.

'`pinoutOffsetX (int)`'
'`pinoutOffsetY (int)`'
: These resources determine the offset in mil of the circuit from the upper left corner of the window when displaying pinout information. Both default to *100*.

'`pinoutTextOffsetX (int)`'
'`pinoutTextOffsetY (int)`'
: The resources determine the distance in mil from the drilling hole of a pin to the location where its name is displayed in the pinout window. They default to *X:50* and *Y:0*.

'`pinoutZoom (int)`'
: Sets the zoom factor for the pinout window according to the formula: scale = 1:(2 power value). Its default value is *2* which results in a *1:4* scale. See also the command-line option *-pz*.

'`printCommand (string)`'
: Default file for printouts. If the name starts with a '`|`' the output is piped through the command. There is no default file or command.

'`raiseLogWindow (boolean)`'
: The log window will be raised when new messages arrive if this resource is set *true*, the default.

'`resetAfterElement (boolean)`'
: If set to *true*, all found connections will be reset before a new element is scanned. This will produce long lists when scanning the whole layout for connections. The resource is set to *false* by default. The feature is only used while looking up connections of all elements. See also the command-line option *-reset, +reset.*

'`ringBellWhenFinished (boolean)`'
: Whether to ring the keyboards bell (the default) when connection searching has finished or not. See also the command-line option *-ring, +ring.*

'`saveCommand (string)`'
: This command is used to save data to a layout file. The filename can be passed by %f in the string. It has to read the data from its standard input. The default command is

    cat - > %f

  See also the command-line option *-sfile.*

'`saveInTMP (boolean)`'
: Enabling this resource will save all data which would otherwise be lost in a temporary file `EMERGENCY_NAME`. *%i* is replaced by the process ID. As an example, loading a new layout when the old one hasn't been saved would use this resource. See also the command-line option *-save, +save.*

'`saveLastCommand (boolean)`'
: Enables the saving of the last entered user command. The option is *disabled* by default. See also the command-line option *-s, +s.*

'`selectedColor (color)`'
: Defines the color to be used to display selected objects. The default is set to *XtDefaultForeground.*

'`useLogWindow (boolean)`'
: Several subroutine send messages to the user if an error occurs. This resource determines if they appear inside the log window or as a separate dialog box. See also the resource *raiseLogWindow* and the command-line option *-loggeometry.* The default value is *true.*

'`viaColor (color)`'
: Defines the color to be used to display vias. The default is set to *XtDefault-Foreground.*

'`viaThickness (dimension)`'
'`viaDrillingHole (dimension)`'
: The initial thickness and drilling hole of new vias. The values must be in the range [MIN_PINORVIASIZE, MAX_PINORVIASIZE] with at least

MIN_PINORVIACOPPER mil of copper. The default thickness is *40 mil* the default drilling hole *20*.

'`size (<width>x<height>)`'
: Determines the maximum size of a layout. The default is *DEFAULT_SIZE*.

'`volume (int)`'
: The value is passed to `XBell()` which sets the volume of the X speaker. The values range is -100..100 and it defaults to the maximum volume of *100*.

'`zoom (int)`'
: The initial value for output scaling is set according to the following formula: scale = 1:(2 power value). It defaults to *3* which results in an output scale of *1:8*.

Refer also to Chapter 4 [Command-Line Options], page 14.

5.2 Actions

All user accessible commands can be bound to almost any X event. No default binding for commands is done in the binaries so it's vital for the application that at least a systemwide application resource file exists. This file normally resides in the *X11/lib/app-defaults* directory and is called *Pcb*. The bindings to which the manual refers to are the ones as defined by the shipped resource file.

Take special care about translations related to the functions keys and the pointer buttons because most of the window managers use them too. Change the file according to your hardware/software environment. You may have to replace all occurances of *baseTranslations* to *translations* you use a `X11R4` server.

Passing *Object* as an argument to an action routine causes the object at the cursor location to be changed, removed or whatever. *SelectedObjects* will handle all selected and visible objects.

'`Bell()`' Rings the bell of your display.

'`ChangeSize(Object, delta)`'
'`ChangeSize(SelectedLines|SelectedPins|SelectedVias, delta)`'
: To change the size of an object you have to bind these action to some X event. All combinations of argument and value, positive or negative, change the size of one object at the cursor position by incrementing it's current size by *delta*. Passing *Selected...* changes the size of all selected and visible objects. Text objects can be used too. Default:

```
None<Key>s:   ChangeSize(Object, 5)
!Shift<Key>s: ChangeSize(Object, -5)
```

'`Change2ndSize(Object, delta)`'
'`Change2ndSize(SelectedPins|SelectedVias, delta)`'
: This action routine changes a second object related size. Right now only the drilling hole of pins and vias is available. Default:

```
!Mod1<Key>s:       Change2ndSize(Object, 5)
!Mod1 Shift<Key>s: Change2ndSize(Object, -5)
```

'`ChangeName(Object)`'
'`ChangeName(Layer|Layout)`'

Changes the name of the visible object at the cursor location. A text objects doesn't have a name therefore it's string is changed. For elements always the visible name is changed. See *Display(CanonicalName|NameOnPCB)* for details. Passing *Layer* changes the current layers name. Default:

```
None<Key>n: ChangeName(Object)
```

'`Command()`'

Calling *Command()* pops up an input line at the bottom of the window which allows you to enter commands. The dialog ends when *None<Key>Return* to confirm or *None<Key>Escape* to abort is entered. Default:

```
<Key>colon: Command()
```

'`Connection(Find)`'
'`Connection(ResetFoundLinesAndRectangles|ResetFoundPinsAndVias|Reset)`'

The *Connection()* action is used to mark all connections from one pin, line or via to others. The *ResetFoundLinesAndRectangles, ResetFoundPinsAndVias* and *Reset* arguments can be used to reset all marked lines and rectangles, vias and pins or all of them. The search starts with the pin or via at the cursor position. All found objects are drawn with the color defined by the resource *connectedColor*. See also *Display(NameOnPCB|CanonicalName)*. Default:

```
!Shift<Key>c: Connection(Reset)
None<Key>f:   Connection(Find)
```

'`Display(CanonicalName|NameOnPCB)`'
'`Display(Grid|Toggle45Degree|ToggleGrid)`'
'`Display(Center|ClearAndRedraw|Redraw)`'
'`Display(Pinout)`'

This action routines handles some output related settings. It is used to center the display around the cursor location and to redraw the output area optionally after clearing the window. Centering is done with respect to the *grid* setting. Displaying the grid itself can be switched on and off by *Grid* but only if the distance between to pixels exceeds MIN_GRID_DISTANCE pixels. `Pcb` is able to handle two names of an element. The first one, the canonical name, should be used to describe it's electronical function (eg MC68040), whereas the second one should be used to inform the user about the functionality (eg CPU). The *Display()* action selects which of the two names is displayed and used when generating connection lists. If *ToggleGrid* is passed `Pcb` changes between relative ('r' in the statusline) and absolute grid (an 'a'). Relative grid means that the cursor position where the last change of the grid setting occured is used as the grid origin, (0,0) is used in case of absolute grid. Passing *Pinout* displays the pinout of the element at the current cursor location. Default:

```
None<Key>c: Display(Center)
None<Key>d: Display(Pinout)
None<Key>r: Display(ClearAndRedraw)
```

'`Load(ElementToBuffer|Layout|LayoutToBuffer)`'
: This routine pops up a fileselect box to load layout or element data. The passed filename for layout data is saved and can be reused for saving it. *ElementToBuffer* and *LayoutToBuffer* load the data into the current buffer. No defaults.

'`Mode(Copy|Line|Move|None|PasteBuffer|Polygon)`'
'`Mode(Remove|Rectangle|Text|Via)`'
'`Mode(Notify)`'
'`Mode(Save|Restore)`'
: Switches to a new mode of operation. The active mode is displayed by a thick line around the matching mode selector button. Most of the functionality of `Pcb` is implemented by selecting a mode and calling *Mode(Notify)*. The arguments *Line*, *Polygon*, *Rectangle*, *Text* and *Via* are used to create the appropriate object whenever *Mode(Notify)* is called. Some of them like *Polygon* need more than one call for one object to be created. *Save* and *Restore* are used to temporarily save the mode, switch to another one, call *Mode(Notify)* and restore the saved one. Have a look to the application resource file for examples. *Copy* and *Move* modes are used to change an objects location and, optionally, to create a new one. The first call of *Mode(Notify)* attaches the object at the pointer location to the crosshair whereas the second one drops it to the layout. Passing *PasteBuffer* attaches the contents of the currently selected buffer to the crosshair. Each call to *Mode(Notify)* pastes this contents to the layout. *Mode(None)* switches all modes off. Default:

```
<Key>Escape:           Mode(None)
<Key>space:            Mode(Notify)
None<Key>BackSpace:    Mode(Save) Mode(Remove) Mode(Notify) Mode(Restore)
None<Key>F1:           Mode(Via)
None<Key>F2:           Mode(Line)
None<Key>F3:           Mode(PasteBuffer)
None<Key>F4:           Mode(Rectangle)
None<Key>F5:           Mode(Text)
None<Key>F6:           Mode(Polygon)
None<Btn1Down>:        Mode(Notify)
!Shift Ctrl<Btn1Down>: Mode(Save) Mode(Remove) Mode(Notify) Mode(Restore)
None<Btn2Down>:        Mode(Save) Mode(Move) Mode(Notify)
None<Btn2Up>:          Mode(Notify) Mode(Restore)
!BTNMOD<Btn2Down>:       Mode(Save) Mode(Copy) Mode(Notify)
!BTNMOD<Btn2Up>:         Mode(Notify) Mode(Restore)
```

'`MovePointer(delta_x, delta_y)`'
: With this function it's possible to move the crosshair cursor by using the cursor keys. The X servers pointer follows because the necessary events are generated by `Pcb`. All movements are performed with respect to the currently set grid value. Default:

```
None<Key>Up:      MovePointer(0, -1)
!Shift<Key>Up:    MovePointer(0, -10)
None<Key>Down:    MovePointer(0, 1)
!Shift<Key>Down:  MovePointer(0, 10)
```

```
None<Key>Right:   MovePointer(1, 0)
!Shift<Key>Right: MovePointer(10, 0)
None<Key>Left:    MovePointer(-1, 0)
!Shift<Key>Left:  MovePointer(-10, 0)
```

'`New()`'
Clear the current layout and starts a new one after entering its name. Refer to the resource *backup* for more information. No defaults.

'`PasteBuffer(AddSelected|Clear|1..MAX_BUFFER)`'
'`PasteBuffer(Rotate, 1..3)`'
This action routine controls and selects the pastebuffer as well as all cut-and-paste operations. Passing a buffer number selects one out of 1..MAX_BUFFER. The statusline is updated with the new number. *Rotate* performs a number of 90 degree counter clockwise rotations of the buffer contents. *AddSelected* as first argument copies all selected and visible objects into the buffer. Passing *Clear* removes all objects from the currently selected buffer. Refer to Section 2.3.6 [Pastebuffer], page 10, for examples. Default:

```
!Ctrl<Key>x:       PasteBuffer(Clear) PasteBuffer(AddSelected)
                   Mode(PasteBuffer)
!Shift Ctrl<Key>x: PasteBuffer(Clear) PasteBuffer(AddSelected)
                   RemoveSelected() Mode(PasteBuffer)
!Shift<Key>1:      PasteBuffer(1)
!Shift<Key>2:      PasteBuffer(2)
!Shift<Key>3:      PasteBuffer(3)
!Shift<Key>4:      PasteBuffer(4)
!Shift<Key>5:      PasteBuffer(5)
None<Key>F3:       Mode(PasteBuffer)
```

'`Polygon((Close|InsertPoint|PreviousPoint|RemovePoint)`'
Polygons need a special action routine to make life easier. Calling *Polygon(PreviousPoint)* resets the newly entered corner to the previous one. *Close* creates the final segment of the polygon. This may fail if clipping to 45 degree lines is switched on. The two other arguments insert a new corner into the segment with the lowest distance from the crosshair position or remove a corner if there are at least 3 others left. Clipping isn't enforced. Both of them only work while in polygon-mode. Default:

```
None<Key>Insert:        Mode(Save) Mode(Polygon) Polygon(InsertPoint)
                        Mode(Restore)
!Shift Ctrl<Key>Insert: Mode(Save) Mode(Polygon) Polygon(RemovePoint)
                        Mode(Restore)
None<Key>p:             Polygon(PreviousPoint)
!Shift<Key>p:           Polygon(Close)
```

'`Print(Layout|Package, PostScript)`'
Pops up a print control box which lets you select some stuff like scaling, colored output, mirroring and rotating. Passing *Layout* prints all visible layers, *Package* only elements and vias. Only `PostScript` is supported right now. The output can be send to a postprocessor by starting the filename with `|`. No defaults.

'`Quit()`' Quits the application after confirming the operation. Default:

```
<Message>WM_PROTOCOLS: Quit()
```

'`RemoveSelected()`'
: This routine removes all visible and selected objects. No defaults.

'`Save(Layout|LayoutAs)`'
'`Save(AllConnections|AllUnusedPins|ElementConnections)`'
: Passing *Layout* saves the layout using the file from which it was loaded or, if it is a new layout, calls *Save(LayoutAs)* which queries the user for a filename. *AllConnections*, *AllUnusedPins* and *ElementConnections* start a connection scan and save all connections, all unused pins or the connections of a single element to a file. No defaults.

'`Select(All|Block|Connection|ToggleObject)`'
: Toggles either the selection flag of the object at the crosshair position (*ToggleObject*) or selects all visible objects, all inside a rectangle or all objects that have been found during the last connection scan. Default:

```
None<Btn3Down>:  Select(ToggleObject)
!BTNMOD<Btn3Down>: Mode(Save) Mode(None) Select(Block)
!BTNMOD<Btn3Up>:   Select(Block) Mode(Restore)
```

'`SetValue(Grid|LineSize|ViaDrillingHole|ViaSize|Zoom, value)`'
: Some internal values can be changed online by this function. The first parameter specifies which data has to be changed. The other one determines if the resource is set to the passed value, if *value* is specified without sign, or increments/decrements if it's specified with plus or minus sign. The function doesn't change any existing object only the initial values of new objects. Use the *ChangeSize()* and *Change2ndSize()* to change existing objects. Default:

```
None<Key>g:        SetValue(Grid, +5)
!Shift<Key>g:      SetValue(Grid, -5)
None<Key>l:        SetValue(LineSize, +5)
!Shift<Key>l:      SetValue(LineSize, -5)
None<Key>v:        SetValue(ViaSize, +5)
!Shift<Key>v:      SetValue(ViaSize, -5)
!Mod1<Key>v:       SetValue(ViaDrillingHole, +5)
!Mod1 Shift<Key>v: SetValue(ViaDrillingHole, -5)
None<Key>z:        SetValue(Zoom, -1)
!Shift<Key>z:      SetValue(Zoom, +1)
```

'`Undo()`'
'`Undo(ClearList)`'
: The implemented unlimited undo feature of `Pcb` allows you to recover from the following operations: *change name*, *copy*, *move* and *remove*. Calling *Undo()* without any parameter recovers from the last of these operations. *ClearList* is used to release the allocated memory. This function is called whenever a new layout is started or loaded. Default:

```
None<Key>u:        Undo()
!Shift Ctrl<Key>u: Undo(ClearList)
```

'`Unselect(All|Block|Connection)`'

Unselects all visible objects, all inside a rectangle or all objects that have been found during the last connection scan. Default:

```
!Shift Mod1<Btn3Down>: Mode(Save) Mode(None) Unselect(Block)
!Shift Mod1<Btn3Up>:   Unselect(Block) Mode(Restore)
```

5.3 Default Translations

This section covers all default translations of key and button events as defined in the shipped default application resource file. Most of them have already been listed in Section 5.2 [Actions], page 20. `Pcb` makes use of a nice `X11` feature; calling several action routines for one event.

'`!Mod1 Ctrl<Key>Left:`'
'`!Mod1 Ctrl<Key>Right:`'
'`!Mod1 Ctrl<Key>Up:`'
'`!Mod1 Ctrl<Key>Down:`'

Scroll one page in one of the four directions.

'`None<Key>Left:, !Shift<Key>Left:`'
'`None<Key>Right:, !Shift<Key>Right:`'
'`None<Key>Up:, !Shift<Key>Up:`'
'`None<Key>Down:, !Shift<Key>Down:`'

Move crosshair either 1 or 10 points in grid.

'`None<Key>Return:`'

Finished user input, selects the 'default' button of dialogs.

'`None<Key>Escape:`'

Mode(Reset), aborts user input, selects the 'abort' button of dialogs or resets all modes.

'`None<Key>BackSpace:`'
'`!Shift<Key>BackSpace:`'
'`!Shift Ctrl<Btn1Down>:`'

The object at the cursor location is removed by *None<Key>BackSpace* or *!Shift Ctrl<Btn1Down>* whereas *!Shift<Key>BackSpace* also removes all other objects that are connected to the one.

'`None<Btn2Down>, Btn2<Motion>, None<Btn2Up>:`'
'`!BTNMOD<Btn2Down>, Btn2<Motion>, !BTNMOD<Btn2Up>:`'

The first sequence moves the object or element name at the cursor location. The second one copies the objects. This function isn't available for element names.

6 File Formats

All files used by `Pcb` are read from the standard output of a command or written to the standard input of one as plain 7 bit ASCII. This makes it possible to use any editor to change the contents of a layout file. It is the only way for element or font description files to be created. To do so you'll need to study the example files `circuits/*` and `FONTFILENAME` which are shipped with `Pcb`. For an overview refer to Chapter 1 [Intro], page 3.

The following sections provide the necessary information about the syntax of the files. The mentioned commands can add almost any additional functionality you may need. Examples are compressed read and write access as well as archives. The commands themselves are defined by the resources *elementCommand*, *fileCommand*, *fontCommand* and *saveCommand*. You should be aware that the commands are not saved together with the data. The advantage is that a layout file holds all information independent of the other files.

One thing in common to all files is that they can include comments, newlines, and carriage-returns at any place except quoted strings.

6.1 Basic Types

Here are the basic type definitions used in the other sections of this chapter.

```
LayoutName      = Name
CanonicalName   = Name
DeltaAngle      = Number
DrillingHole    = Number
Flags           = Number
FontPosition    = Number
Grid            = Number
GridOffsetX     = Number
GridOffsetY     = Number
Group           = decimal [,decimal]...
GroupString     = """ Group [:Group]... """
Height          = Number
LayerNumber     = Number
Name            = quoted_string
Number          = decimal | hex
Spacing         = Number
StartAngle      = Number
SymbolID        = Number | charconst
Thickness       = Number
TextData        = quoted_string
TextFlags       = Flags
TextScale       = scale
TextX           = Number
TextY           = Number
Width           = Number
X               = Number
X1              = Number
```

```
X2               = Number
Y                = Number
Y1               = Number
Y2               = Number
charconst        = "’" <any character> "’"
comment          = "#" {<any character up to a newline>}...
decimal          = [0-9]+
direction        = [0-3]
hex              = 0x[0-9a-fA-F]+
scale            = [1-<positive integer>]
quoted_string    = """ <anything except \n and \r> """
zoom             = [0-MAX]
```

6.2 Layout File Format

The layout file describes a complete layout including symbols, vias, elements and layers with lines, rectangles and text. This is the most complex file of all.

```
File             = Header Font PCBData
Header           = PCBName [GridData] [CursorData] [PCBFlags] [Groups]
PCBName          = "PCB(" Name Width Height ")"
GridData         = "Grid(" Grid GridOffsetX GridOffsetY ")"
CursorData       = "Cursor(" X Y zoom ")"
PCBFlags         = "Flags(" Flags ")"
Groups           = "Groups(" GroupString ")"
Font             = {FontData}...
FontData         = {Symbol}...
Symbol           = "Symbol(" SymbolID Spacing ")"
                       "(" {SymbolData}... ")"
SymbolData       = {SymbolLine}...
SymbolLine       = "SymbolLine(" X1 Y1 X2 Y2 Thickness ")"
PCBData          = {Via | Layer | Element}...
Via              = "Via(" X Y Thickness DrillingHole Name Flags ")"
Element          = "Element(" Flags CanonicalName LayoutName \
                       TextX TextY direction scale TextFlags")"
                       "(" {ElementData}... [Mark] ")"
ElementData      = {ElementLine | Pin | ElementArc }...
ElementLine      = "ElementLine(" X1 Y1 X2 Y2 Thickness ")"
Pin              = "Pin(" X Y Thickness DrillingHole Name Flags ")"
ElementArc       = "ElementArc(" X Y Width Height
                       StartAngle DeltaAngle Thickness ")"
Mark             = "Mark(" X Y ")"
Layer            = "Layer(" LayerNumber Name ")"
                       "(" {LayerData}... ")"
LayerData        = {Line | Polygon | Text}...
Line             = "Line(" X1 Y1 X2 Y2 Thickness Flags")"
Polygon          = "Polygon(" Flags ")" \
                       "(" {Points}... ")"
```

```
Points          = "(" X Y ")"
Text            = "Text(" X Y direction scale TextData Flags")"
```

'`PCBName`' is used to define the layouts name which is independent of it's filename. It is displayed in the lower left corner of the main window.

'`GridData`'
is optional and is used to save the grid setting and offset that was set at the time the layout was saved.

'`CursorData`'
is also an optional parameter to save the last cursor location and zoom value. The real zoom factor is calculated by scale = 1:(2 power value).

'`PCBFlags`'
determine how to draw lines and which name of the elements should be displayed.

bit 5:	display canonical element names if set
bit 7:	use absolute grid if set
bit 8:	don't clip lines to 45 degrees

'`Groups`' Layergroups are saved by using this optional parameter. The only way of changing them is to use an editor and alter the appropriate line.

'`Symbol`' See the description of font files in this chapter.

'`Via`' Vias are always connected to all layers which also means that they are one logical level ahead of them. They are defined by position, size, name and by some flags.

bit 0:	always clear
bit 1:	always set
bit 2:	set if via was found during a connection search

'`Element`' See the description of element files in this chapter.

'`Layer`' A layer is the central object from the users point of view. It holds all connections and all text objects. Up to MAX_LAYER can be used individually. Its number, starting with 1, and its name are read as arguments.

'`Line`' All lines are identified by their start and endpoints together with their thickness and some flags. They have to fit a 45 degree scheme.

bit 2:	set if line was found during a connection search
bit 6:	line has been selected

'`Polygon`' used to fill a larger area with '`copper`'. The coordinates specify the the corners. The flags are:

bit 2:	set if polygon was found during a connection search
bit 6:	polygon has been selected

'`Text`' You may use text objects to add information to your board. An example would be naming a connector or marking pin one of it. The position marks the lower left corner of the string which is also a fixpoint for rotations. The directions are independent to the ones of lines. They are counted from zero to three with a meaning of zero to 270 degree rotations counter-clockwise. One bit of the flags field is used to enable mirroring. The scaling value is a positive integer which determines a zoom factor in percent.

bit 3:	if set, use mirroring to draw the text string
bit 6:	the text has been selected

6.3 Element File Format

Element files are used to describe one component which can then be used several times within one or more layouts. You will normally split the file into two parts, one for the pinout and one for the package description. Using `m4` allows you to define pinnames as macros in one file and include a package description file which evaluates the macros. See the resource *elementCommand* for more information.

Going this way makes it possible to use one package file for several different circuits. See the sample files `dil*`.

The lowest x and y coordinates of all subobjects of an element are used as an attachment point for the crosshair cursor of the main window.

```
File           = {Element}...
Element        = "Element(" Flags CanonicalName LayoutName \
                     TextX TextY direction scale TextFlags")"
                     "(" {ElementData}... [Mark] ")"
ElementData    = {ElementLine | Pin | ElementArc }...
ElementArc     = "ElementArc(" X Y Width Height
                     StartAngle DeltaAngle Thickness ")"
ElementLine    = "ElementLine(" X1 Y1 X2 Y2 Thickness ")"
Mark           = "Mark(" X Y ")"
Pin            = "Pin(" X Y Thickness DrillingHole Name Flags ")"
```

‘`Element`’ Objects of type element are determined by two names, a canonical and a layout name, the according text position, its direction counted from 0 to 3 (n * 90 degrees counter-clockwise) and the data.

‘`Flags`’ The flag field determines the state of an element. The bit values are:

bit 6: element has been selected

‘`TextFlags`’
‘`scale`’
‘`direction`’
See the description of text object earlier in this chapter.

‘`ElementLine`’
A line by its start and endpoint and its size. Only 45 degree lines are supported.

‘`ElementArc`’
Defines an arc by its center, width, height, startangle, its length in degrees and its size. Remember that the y axis on the screen grows downwards.

‘`Mark`’ is just a hint to make positioning easier. The crosshair will be positioned here. Its center is passed as the two arguments.

‘`Pin`’ A pin is very similar to a via except that it can't be removed independent of the element to which it belongs. They are defined by position, size, name and by some flags.

bit 0:	always set
bit 1:	always clear
bit 2:	set if pin was found during a connection search
bit 6:	pin has been selected

6.4 Font File Format

A number of user defined symbols are called a font. There is only one per layout. All symbols are made of lines. See the file `FONTFILENAME` as an example.

The lowest x and y coordinates of all lines of a font are transformed to (0,0).

```
File            = Font
Font            = {FontData}...
FontData        = {Symbol}...
Symbol          = "Symbol(" SymbolID FontPosition ")"
                       "(" {SymbolData}... ")"
SymbolData      = {SymbolLine}...
```

‘`Symbol`’ The two arguments are the ASCII code of the symbol and it's distance to the next symbol. Undefined symbols are drawn as filled rectangles. The ASCII code can be passed as a character constant or a hex value.

‘`SymbolLine`’
The symbol data itself is made up of several entries of type *SymbolLine*. Be aware that only 45 degree lines are supported.

Appendix A Installation and Troubleshooting

Compiling and installing the package should be straightforward. If any problems occur, please contact the author (Thomas.Nau@rz.uni-ulm.de) to find a solution and include it into the next release.

A.1 Compiling and Installing

This section covers the steps that are necessary to compile the package.

A.1.1 Editing the Imakefile

Most `X11` related options are automatically covered by `imake` which is called from `xmkmf`. The ones special to `Pcb` which have to be edited in `Imakefile` are:

‘`INFOLIBDIR`’
: has to be set to the directory where your GNU info files are located in.

‘`PCBLIBDIR`’
: is the path of a directory where the font files ... go to.

‘`CIRCUITDIR`’
: a directory where electrical circuits go to.

‘`PACKAGEDIR`’
: a directory where packages go to.

‘`EXTRA_INCLUDES`’
: Some systems do not have the Athena Widget include files in their normal place as configured by `X11`s config files. Define this like

```
EXTRA_INCLUDES = -I/usr/openwin/share/include
```

This is probably true for Suns which use `OpenWindows`.

‘`EXITCALL`’
: The symbol *EXITCALL* should be defined according to the *call on exit* functions supported by your system. There are three choices:

```
EXITCALL = -DHAS_ATEXIT    if atexit() is supported (SYSV)
EXITCALL = -DHAS_ON_EXIT   if on_exit() and no atexit() is supported
EXITCALL =                 if none of them is supported
```

Please check your manpages for details.

‘`SYS_LIBRARIES`’
: This symbol is used to pass additional libraries to the linker. The only additional libraries that are used are the math and lex library.

‘`PATCHES`’
: This symbol is passed to the compiler. Use it to define additional stuff. Add *-DNEED_STRDUP* if your system does not have a strdup() library function.

If you have to make system dependent changes please include them into a *#ifdef Architecture ... #endif* construct and mail a copy to the author (Thomas.Nau@rz.uni-ulm.de).

Now run `xmkmf -a` to create the `Makefile` and do some stuff like `make depend`. This should finish without any problems beside some systems that complain about missing include files. Don’t care about that at this time, the package should compile without any problems.

A.1.2 Manuals

After `xmkmf -a` has created the new `Makefile` you are able to create the manpages, the application resource, the info file, the `TeX` output and a reference card by executing

```
        make pcb.man
        make Pcb.ad
        make pcb.info
        make pcb.ps        or  make pcb.dvi
    make refcard.ps    or  make refcard.dvi
```

You'll need `TeX`, `texindex` and, if you want `PostScript`, `dvips` to build the manuals. Preformatted documentation for the default configuration is available from the `./doc` directory. A simple *make* builds the binary, manpage and the resource file. Get yourself a printed copy to make life easier. `TeX-3.0` failed, 3.14 worked just fine.

A.1.3 Compiling the Package

After reaching this point, it's time for `make`. It should build the program, the application resource file and the man-page without any errors. If it doesn't refer to Section A.2 [problems], page 32.

You have to be root to install the package or at least a user with the appropriate right on some `X11` directories. Set umask to *022* else the will not be found because the directory isn't world readable.

`make install` and `make install.man` install the program, the fonts, the application default resource file, all element and package files as well as the manpage to the `X11` directories. `make install.info` does the same for the GNU info file.

A.2 Troubleshooting

There are some problems that are already known. Most of them are related to missing parts of a standard `X11` distribution. Some others are caused by 3rd party applications like X servers. To make this list more complete please mail your problems and, if available, solutions to the author. The mail address can be found at the beginning of this chapter. In any case, read Section A.2.7 [X11], page 34.

By the way, you MUST HAVE AN ANSI COMPILER to make it work. The latest versions of `bison` and `flex` are highly recommended as a replacement for `yacc` and `lex`. Problems with syntax errors while parsing the demo files are related to old `lex` code.

If the shell script `create_sed_script.sh` fails with an error of `awk` check your system for `nawk` or get the GNU `gawk`. See the script for details.

Another source of problems are older versions of `flex` and `bison`. `Pcb` definitely works with `flex-2.4.7` and `bison-1.22` or later. The problems will result in a *syntax error* while parsing files. You will have to add *-lfl* to the SYS_LIBRARIES identifier in `Imakefile`. See also Section A.2.6 [FreeBSD], page 34.

A.2.1 HP Series 300, 700 and 8x7 with X-Terminal (R5)

You have to install several `X11` tools (`imake`) and include files or, better, install a complete `X11R5` release. HP doesn't support the Athena Widgets so all header files and libraries are missing. They also don't ship the ANSI compiler with the normal OS release so you have to buy one or use GCC.

Beside that, `Pcb` has been successfully tested on these platforms with `HPUX 9.0[0134]` and `X11R5`.

A.2.2 Sun Sparc Station II

There are no known problems with Sun machines if they use `X11R5` instead of `OpenWindows`. `Pcb` compiled successfully with a Sparc Station 2, Station 5 and 10 running `BSD-4.1.3` or `Solaris-2.3`.

For problems with `OpenWindows` refer to Section A.2.7 [X11], page 34.

If `xmkmf` is missing, try

```
/usr/openwin/bin/imake -DUseInstalled -I/usr/openwin/lib/config
make Makefile
make includes
make depend
```

instead. I got it compiled with `gcc-2.3.3` but the linker complained about missing symbols

```
ld: Undefined symbol
    _get_wmShellWidgetClass
    _get_applicationShellWidgetClass
```

The problem can be related to the mixed `OpenWindows - X11R5` environment which is installed on the test machine. Anyway the code was executable and I haven't got a core yet.

A.2.3 Silicon Graphics Indigo

`Pcb` has been tested on a Iris Indigo, `IRIX-4.0.5`, with a `X11R4` server. There are no problems beside some additional compiler flags. `X11R5` was used to compile the program. For known problems with `X11R4` see Section A.2.7 [X11], page 34. Check `Imakefile` too.

A.2.4 SCO Unix

John DuBois <spcecdt@deeptht.armory.com> wrote:

```
SCO-ODT-3.0 requires the latest version of tls003, the Athena
widget library (available from sosco.sco.com). The main problems
I have encountered are that it core dumps fairly often, especially
while loading/dropping elements...
```

I'll see what I can do as soon as I have access to a SCO system.

A.2.5 Linux

Since the `X11` version of `Pcb` has been developed on a Linux system with `XFree-2.1` (S3 server) there are no known problems. The older problem about an ATI VGA card that locks up the server has been tracked to the board itself. The S3 server of `XFree-3.1` works fine too.

A.2.6 FreeBSD and NetBSD

If `Pcb` complains about syntax errors even in the demo files get rid of your `lex` and `yacc` implementation. Replace them by GNU `flex` and `bison`. Don't forget to change the SYS_LIBRARIES in `Imakefile` from *-ll* to *-lfl*. You also have to define

YACC = bison -y
LEX = flex

A.2.7 Problems related to X11

There are a some problems related to `X11R4` or systems derived from `X11` like `OpenWindows`. See Section A.2.2 [Sun], page 33. You have at least to change all occurances of *baseTranslations* in the resource files to *translations* if you are using a R4 server. Have a look to the `X11R5` *Intrinsics* manual for details.

The panner widget (print dialog box) appears only in release 5 and later. It really simplifies adjusting the offsets. With earlier releases the printout will always appear in the upper left corner with respect to the set margins.

You may have some problems in a mixed `X11 OpenWindows` environment. If you want to try it anyway you have to add an additional path for include files and define another symbol in `Imakefile`,

```
EXTRA_INCLUDES = -I/usr/openwin/include
```

`Pcb` has been tested successfully with `X11R6` under Linux 1.1.59.

A.2.8 Problems related to TeX

If your `TeX` installation complains about a missing `texinfo.tex` file copy the one included in this release to your `TeX` macro directory. Be aware that there are probably newer versions of this file available from some ftp sites. `TeX-3.0` failed, `TeX-3.14` worked just fine. Check our ftp server *ftp.medizin.uni-ulm.de* for a prebuild version of the manuals.

Index of Resources

Index of Actions, Commands and Options

Index of Concepts

Table of Contents