BEAR MANUAL

by

Wei-Ming Dai, Margaret Marek-Sadowska,
Benjamin Chen, Massoud Pedram, Sherry Solden

Memorandum No. UCB/ERL M89/36

12 April 1989

BEAR MANUAL

by

Wei-Ming Dai, Margaret Marek-Sadowska,
Benjamin Chen, Massoud Pedram, Sherry Solden

Memorandum No. UCB/ERL M89/36

12 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BEAR MANUAL

by

Wei-Ming Dai, Margaret Marek-Sadowska,
Benjamin Chen, Massoud Pedram, Sherry Solden

Memorandum No. UCB/ERL M89/36

12 April 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

BEAR Manual

Table of Contents

Introduction

BEAR is a second generation macrocell-based layout system being developed at U. C. Berkeley. The system takes advantage of our experience with BBL (Berkeley Building-Block Layout System [1]) and feedback from industry; our goal is to provide automatic and interactive features to lay out a chip in both top-down and bottom-up physical design environments. This system has two unique features: a new architecture which employs strong interaction between placement and routing and a dynamic and efficient data representation which unifies topological and geometrical information.

Although placement and routing are interdependent, they have historically been approached separately because of the complexity of computation. Even with very sophisticated placement and routing techniques, a system will not guarantee an appropriate layout if the placement and global routing solutions are mismatched. We believe that floorplanning/placement should be refined more and more (with possible topological changes) as routing proceeds and global routing should be updated incrementally.

Placement defines the capacity of the routing area around the blocks; global routing defines the density (net assignment) of the routing area. Considering the detailed routing, the desirability of a particular global routing on a given placement depends on the degree of the match of capacity and density. After placement and global routing we can change the density by global re-routing (at present this is done manually) or we can change the capacity by global spacing (global compaction or decompaction). In order to achieve high density in the final layout, we iterate these two operations to obtain a satisfactory match of the capacity and density of the routing area before detailed routing. During global spacing, global routing is updated incrementally. A dynamic data representation which unifies topological and geometrical information is used to achieve an efficient implementation of these difficult operations The global spacing organizes the interaction between placement and global routing in a dynamic way. Going one step further, we can optimize the shapes of the soft blocks after global routing. At that point, the global routing is already done, so we can make use of this information in our shape optimizer.

After making a considerable effort to interact placement with global routing, BEAR allows an additional opportunity to refine the placement during local routing. To make such refinement robust and efficient, a feasible routing order is crucial. In a feasible routing order, a new channel can be expanded or contracted when it is being routed without destroying the previously routed channels. In this way, routing can be completed without iteration. Rather than restricting the floor plans or placements to slicing structures (finding a feasible routing order in such special case is trivial) as most systems do, BEAR provides a feasible routing order for non-slicing structures. A highly efficient hierarchical global router is used in conjunction with a global spacer to match the channel capacity and routing density. An L-shaped channel router together with a regular channel router is used in conjunction with the channel spacer NUTCRACKER for detailed routing. Also recently completed is a ring router which makes a connection to the I/O pads, including power and ground nets.

This manual describes the BEAR system in three sections. In section 1, installation instructions and user interface are given. Also a step-by-step running example is provided to help users get started. Clustering, placement, and shape optimization are described in Section 2. Although this part of the system is fully automated, the set of parameters and graphic display provide controllability and observability. In Section 3, routing and spacing processes are discussed. While we provide a set of operations (or mechanisms), users have the freedom to choose when and how to apply them. The input file format is described in Appendix I. BEAR runs on both color and black-and-white workstations which support the X-window system (currently X10). Users may customize the colors, font styles, and other parameters by setting the *X defaults* file (Appendix 2, p. 45). The BEAR system has been integrated into the OCT framework (Appendix 3, p. 48).

I. Installation and User Interface

A. General Information

1. Hardware requirements.

- Disk Space: about 30 megabytes of disk space is required to compile and run the system.

2. Software requirements.

- X-Window System, version 10 (color or black & white). The system has been tested on a black & white and 6 & 8 plane color monitors.
- UNIX operating system. The system has been tested on Ultrix 2.2 and 4.2 BSD.
- Fortran Compiler (Fortran-77).
- C compiler: cc. The system has not been tested using gcc.
- Optional: OCT library.

3. Notes.

BEAR has been tested on:

- Sun 3/60C running NFS and 4.2 BSD with 16 megabytes of real memory and a CGFOUR display driver.
- DEC microvax GPX/II running Ultrix 2.01 and 2.2 with 8 megabytes of real memory and B&W and 8-plane color displays.

B. Getting Started

This section of the manual gives step-by-step instructions for getting BEAR from the tape into the system.

1. Reading from the tape.

BEAR is stored in the tar format so it can be read by any UNIX-like system. The blocking factor of the tape record is 20, which should be the default on most systems. At any rate, this parameter is already determined when reading in the tapes. To get BEAR onto the system, you should do the following:

a. Load the tape into your tape drive so that it is ready for reading. Be sure the write-protect mechanism on the tape is activated to avoid accidental erasure of media.

b. Make a directory where BEAR will reside. The program will need about 30 megabytes of disk space to compile. A typical command might be:

```
% mkdir /users/bear
```

This directory will be the one used for the rest of this guide.

c. Now you are ready to read from the tape.

```
% tar x /users/bear
```

This process will take a while depending on how fast your system is, so be patient.

d. When the prompt returns, the taping has been finished. Type

```
% cd /users/bear/bear/src
```

and then

```
% ls -F
```

to list the source directories that should be present:

```
bearGrUtils/        gtTree/             rectSlice/
bearMisc/           input/              ringRouter/
cluster/            localRouter/        routeDB/
ezPlot/             localSpacer/        textio/
floorplan/          localVia/           tileMapping/
fpg/                mac/                tileProp/
geoChDecomposer      magicMisc/         tiles/
globalRouter/       octInterface/       topChDecomposer/
globalSpacer/       pgRoute/            utils/
grEditor/           placer/
```

2. *Compiling* BEAR.

To compile the system, some variables must be set describing the environment in which BEAR resides. Also, a few local directories must be installed before proceeding to compile BEAR.

a. The main Makefile must be edited to match the system. This file resides in /users/bear/bear. You can use your favorite editor to do accomplish this task. Find the place in the Makefile where you see:

   BEAR_DIR = /users/bear/bear

   This specifies where the BEAR resides. In this case, the default directory matches the directory previously recreated. If the default differed then */users/bear* would be replaced by the directory specified by the initial **mkdir** command.

b. Search for:

   MACHINEFLAGS = -DVAX

   This variable specifies what type of machine is be used. It can be set to *-DVAX* or *-DSUN2*. If the machine is not a VAX or a Sun 2 then the variable should be set to *-DNEITHER*. Currently, the only machine-dependent code is located in *$BEAR_DIR/src/utils/whence.s.*

c. Quit the editor. Be sure you are in the BEAR directory. In this case, it is */users/bear/bear.* Now type:

   ```
   % make setupdirs
   ```

   This directive installs the directories necessary to support the BEAR compilation and coding environment.

d. The program is ready to be compiled. Type:

   ```
   % make install
   ```

   This command will compile all of the libraries, support modules, and BEAR modules and link them together. The runnable program will reside in */users/bear/bear/src/polarBear* and it will be called *bear*. If changes are made to the BEAR code, the BEAR modules can be recompiled and linked by typing **make installbear** in place of **make install**.

C. User Interface

BEAR uses the X Window System as its interface to the user. Two major types of windows are used along with different types of dialog boxes to converse with the user and display the status of the program. The console window is the main root window where commands are issued. Chip windows graphically display the status and characteristics of the current layout example. Three types of user interface are used. The first is for interactive variables (IV) which allow the user to modify various parameters of the program. The second is an X dialog manager (XDM), which prompts the user for more information when required. The last type of interface is the X deck of cards menu system which is used to allow user commands to be entered by using the menus rather than the keyboard.

1. The console window.

The console window of BEAR is the main area where the user directs program action . Commands are specified in the UNIX tradition. Each command consists of a command name followed by an optional list of arguments usually preceded by a '-'. Each command that has optional arguments recognizes the *-help* option which causes a small summary of all the command options to be printed out in the console window. For primitive editing, a few simple key strokes have been defined. The last word of text on a command line can be deleted by typing **control-w (^W)**, while the entire line may be deleted with a **control-u (^U)**. The console window be closed by typing **control-d (^D).**

Unless a default geometry has been specified in the user's *~/.Xdefaults* (See XDefaults), the program prompts the user to create the console window upon invoking the program. Once the console window has been placed, a prompt is displayed when the program is ready to accept commands. Holding down the middle mouse button on the window will display a menu of commands directly related to the console.

Along the right side of the console window is a scroll bar window. The scroll bar window displays a filled square representing the relative position through the window and the relative amount of the window currently on the screen. Scrolling is controlled by clicking mouse buttons in the scroll bar. The middle button scrolls to a particular spot in the window. This operation causes the screen to scroll so that the center of the scroll bar indicator moves to the current position of the mouse. The other two mouse buttons are used for scrolling down or up some proportion of the screen. The left button causes the screen to scroll so that the line adjacent to the mouse position becomes the top line of the screen. Thus, clicking near the top of the scroll bar scrolls only a couple of lines while a click near the bottom will scroll almost an entire screen. The right button causes the top line of the screen to scroll down to the current position of the mouse.

2. The chip window.

The chip window of BEAR is the main area where the user can view the results of his or her commands. It displays the current layout of the chip using graphical abstractions. Cells are represented by rectilinear shapes. Pins are fixed-size squares usually placed along the inner boundary of cells. Regions of the chip window can be magnified for closer inspection. Other abstractions displayed in the manual are explained in detail below.

3. Interactive variables (IV).

This form of dialog is specifically used to view and edit program variables. The appearance of these dialogs are very distinct. A title describing the current operation is displayed at the top of the dialog. All the interactive variables are shown on a window, one on each row. Each variable is displayed with its description and a region containing its current value.

At any one time, the IV window maintains at most one active edit region where the variable may be changed. All keyboard input anywhere in the IV window will be directed to this region. Edit regions are activated by placing the mouse cursor over an edit region, and either clicking a mouse button or pressing a key. This action is indicated by a cursor (a pointer under a line of text) inside the active edit-region. The user is not allowed to enter more text than there is space in the edit region component. *Changes are accepted only by a carriage return or end-of-file.* The original value of the variable can be restored by typing **control-u (^U)** before accepting any changes. Edit regions are usually denoted by a different color background where the value of the variable is displayed. Edit regions whose background color matches the background color of the entire window are read-only, unless buttons are present.

For integer or floating-point variables, two buttons are provided to change the value of the variable. The "+" button has the following effect:
If the *LEFT* mouse button is pressed, the value of the variable is incremented by 1%, or by one for integer variables.
If the *MIDDLE* mouse button is pressed, the value of the variable is incremented by 10%
If the *RIGHT* mouse button is pressed, the value of the variable is doubled.

The "-" button has similar behavior, but the value of the variable is decremented. Integer variables can be distinguished from floating point variables by the presence of a decimal point. For variables with strings as values (except booleans), the plus and minus buttons advances or reviews through a list of values that the user can choose. For boolean variables, one button is provided for easy toggling of its state. Single buttons are also provided for directing actions such as aborting the dialog.

4. X Dialog Manager (XDM).

This form of dialog is designed to query input from the user. The two major types of components for these dialogs are check boxes and edit regions. A check box appears as a small box with rounded corners. When a button is clicked inside the box, a small check mark is drawn inside the box indicating a choice has been made. Edit regions are similar to those of the interactive variables dialogs. At any one time, each top-level dialog component maintains one active edit-region component. This component is indicated by a cursor (a pointer under a line of text) inside the active edit-region. All keyboard input anywhere in the dialog will be directed to this component. All normal printing characters insert themselves into the edit region at the current cursor location. The user is not allowed to enter more text than there is space in the edit region component. In addition, many character control sequences can be used for basic text editing operations:

Key	*Description*
^A	Move to beginning of the line
^E	Move to the end of the line
^P	Move to the previous line
^N	Move to the next line
^F	Move forward one character
^B	Move backward one character
^H or <del>	Delete the previous character
^D	Delete the next character
^U or ^X	Delete the current line

There are several ways to change the currently active edit region component. First, the user can move the mouse over another edit region field and press a mouse button, thus activating that field. Second, the user can type **<tab>** and **^Q** to move to the next and previous fields respectively. The next and previous fields are determined by the order of creation of edit region components. Typing **<tab>** in the last edit region to be created causes the first edit region created to become active. Similarly, typing **^Q** in the first edit region to be created causes the last edit region created to become active. Finally, typing **^N** in the last line of an edit region is equivalent to typing **<tab>**, and typing **^P** in the first line of an edit region is equivalent to typing **^Q.** As with IV, edit regions are sometimes disabled, disallowing any input to the region. These are indicated by a shaded edit region.

All XDM dialogs consists of a title, "ok" and "abort" buttons. There is one special slider component. This component contains a slider and two buttons to increment or decrement the slider. The value field of the slider may also be edited with the same usage as in the edit region of an IV variable.

5. X Deck of cards Menu System (XMenu).

XMenu is an X Window System Utility Package that implements a 'deck of cards' menu system. XMenu is intended for use in conjunction with Xlib, the C Language X Window System

Interface Library.

In a 'deck of cards' menu system a menu is composed of several cards or panes. The panes are stacked as if they were a fanned-out deck of playing cards. Each of these panes has one or more selections. A user interacts with a 'deck of cards' menu by sliding the mouse cursor across the panes of the menu. As the mouse cursor enters each pane it will rise to the top of the deck and become 'current'. If the current pane is an active pane it will be 'activated', or made available for selection. To indicate this, its background will change from the patterned inactive background to a solid color and the selections on the pane will be activated. If the current pane is not an active pane (a setable state) then it will not be activated. To indicate this its background will continue to be the patterned inactive background and no selections on the pane will be activated. The pane previously containing the mouse will lower (preserving its stacking order). If it was activated it will then become deactivated, its background changing back to the inactive pattern. Because of this action it is not possible to have more than one current pane at any one time. When the mouse cursor enters an active selection in a pane that has been activated then that selection will become activated and be highlighted. If the selection is not active or the pane has not been activated then the selection will not be activated and will not be highlighted. Selection highlighting is accomplished in one of two ways depending upon the state of the user's Xdefaults variables. If 'box' mode highlighting is in effect, the menu selection will be activated by placing a highlight box around the selection as the mouse cursor enters the selection's active region and removing it (deactivating the selection) as the cursor leaves. If 'invert' mode highlighting is in effect, the menu selection will be activated by inverting the background and foreground colors within the selection's active region as the mouse cursor enters it and reinverting them as the cursor leaves.

The purpose of these menus is singular: to present the common console window commands to the novice user, though experienced users may find this way of entering commands easier than entering commands through the keyboard. XMenu's are available in every type of window created by BEAR, but the commands it lists are context-sensitive. The commands displayed are only the commands valid for that particular window. Upon choosing a command, the keyboard abbreviation is echoed into the console window.

D. Sample Layout

This section of the manual describes a typical set of instructions that is used to complete the routing of a chip.

1. Starting the program.

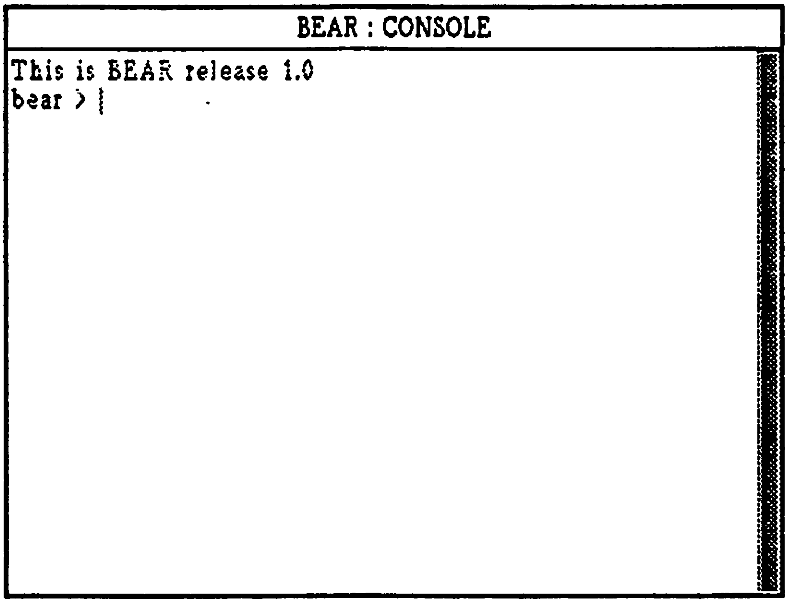
The first step is to start the program:

% bear

Be sure your DISPLAY environment variable is set. If it is not, the user can specify the display on which to run BEAR by typing **bear petrus:0**, where **petrus** is the hostname of the destination machine.

2. Start log file.

At this point the BEAR console window will be created and a prompt displayed. (See Fig. 1.)



```
This is BEAR release 1.0
bear > |
```

Fig. 1.

Now the user should type:

bear > **log sample.log**

This creates a file in your current working directory called *sample.log*. It will contain all messages echoed to the console window from the time the user invoked the command.

3. Read in the placement file for the chip.

Next, type:

bear> **ow -bbl testData/iccad1.r**

The command **ow** stands for open window. A window will be created displaying the cell layout for the *iccad1.r* example, found in the *testData* directory, which has been stored in BBL format. (See Fig. 2.)

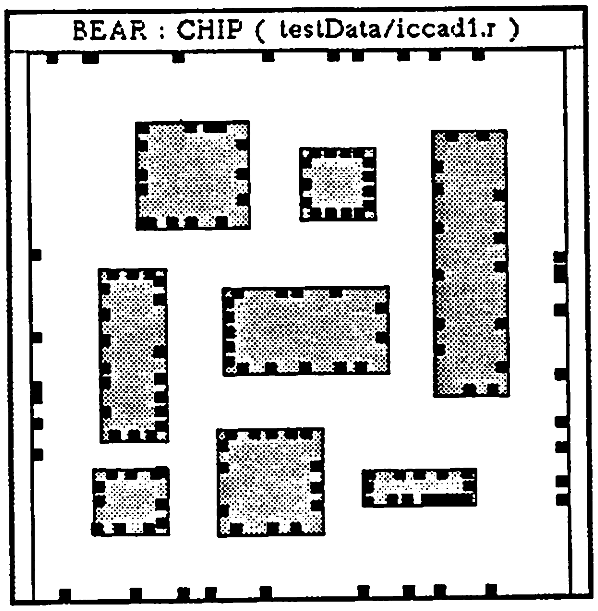


Fig. 2.

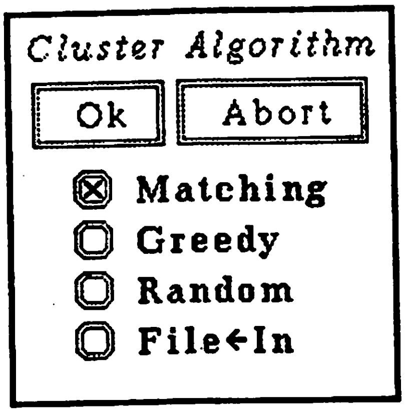
The *testData* directory is in your current working directory in this case. Although not specified directly, one other file must exist in the same path as *iccad1.r*. This file is *iccad1.tech*. An optional file, *iccad1.wt*, used for power and ground sizing should also be in this path. The format and usage of these files are described in detail later in the manual.

4. Create clustering tree.

Now that the initial placement file has been loaded, BEAR is ready to optimize the placement. This step is optional as the user may proceed directly to routing. Usually, the first step in placement optimization is the creation of a *clustering tree*. This task is accomplished by the **cl** command. Remember that the mouse cursor must be in the chip window for this command to be valid.

bear> **cl**

After this command is entered, a dialog box will be created (centered about the current mouse position) since more information is required. The user must specify the type of clustering algorithm. The matching algorithm usually gives good results. Select this algorithm by placing the mouse cursor over the box adjacent to the name and clicking the button. An *x* will mark the choice. (See Fig. 3a.)



[✓ Matching]
Greedy
Random
File←In

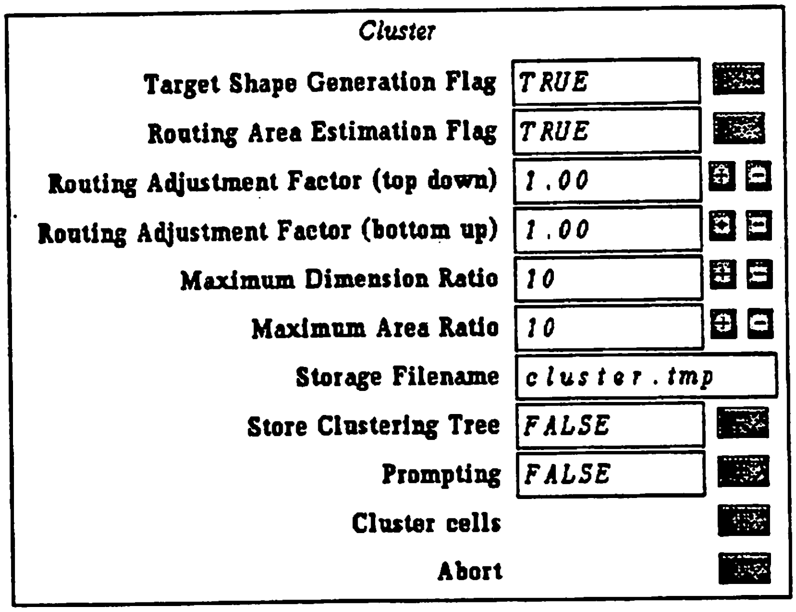
Fig. 3a.

Then click the button over the *Ok* box. *Abort* will close the dialog box and return input to the console window. After the choice has been made, a message will be echoed to the console:

cl -a m

This message is an abbreviation for the command specified by the dialog box. In the future, the user may wish to enter the abbreviation instead of working with the dialog box.

Immediately following the dialog box, an IV (interactive variable) window is created centered around the current mouse position. (See Fig. 3b.)



Cluster

Target Shape Generation Flag	TRUE
Routing Area Estimation Flag	TRUE
Routing Adjustment Factor (top down)	1.00
Routing Adjustment Factor (bottom up)	1.00
Maximum Dimension Ratio	10
Maximum Area Ratio	10
Storage Filename	cluster.tmp
Store Clustering Tree	FALSE
Prompting	FALSE
Cluster cells	
Abort	

Fig. 3b.

Variables are displayed to allow the user to change the default values for the clustering parameters (see section on Clustering and Placement). In general, the user accepts the default values within IV windows. In this case, a click on the button adjacent to the *Cluster cells* label will accept the defaults and begin generating the clustering tree. When the tree has been generated, this message will be displayed in the console window:

Clustering is finished

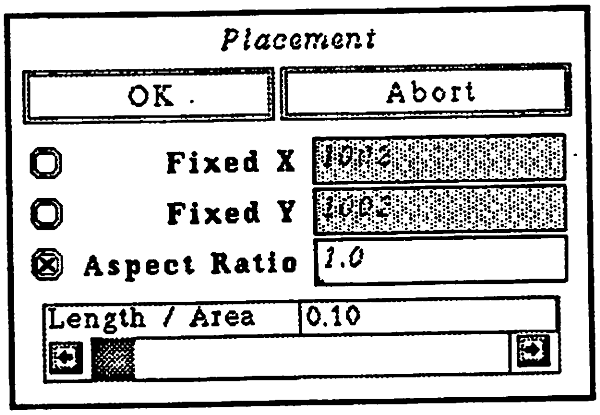
And the prompt will return.

5. Placement of cells.

Now the placement of the cells can be computed. This task is done with the **pl** command.

bear > **pl**

A dialog box will appear with default values in its fields. (See Fig. 4.)



Fixed X 1002
Fixed Y 1002
[✓ Aspect Ratio] 1.0
Length / Area 0.10

Fig. 4.

This particular dialog has some unique features. Boxes along with type-in fields appear next to parameter names. Marking one box disables all the other type-in fields. The disabled options are highlighted by shaded type-in regions, and input is not allowed to these regions. Just below these type-in fields is a slider component. This component is described in the User Interface portion of the manual. In this case, all of the defaults will be accepted by clicking the mouse over *Ok.* As with the cluster command, the equivalent command line sequence is echoed to the console window:

pl -ar 1.0 0.1

At this time, the user is prompted for the lookahead constant.

Cluster level 0

Depth of lookahead [0]:

See the section on Clustering and Placement for details. The default value is printed in the brackets and is accepted by a carriage return. The message:

Placing cells ...

is echoed to console. This portion of the program usually takes some time to finish. The conclusion of placement is signaled by a message:

Placing cells ...done

and the command line returns.

6. Viewing the clustering tree and placement.

The user may view the clustering tree and placement of the cells before accepting them. The command is:

bear > **ow -tf**

The **-tf** option stands for "tree-floorplan." The user will then be prompted to create two windows. Initially, these windows display the root node of the clustering tree and its corresponding placement. (See Fig. 5a.)

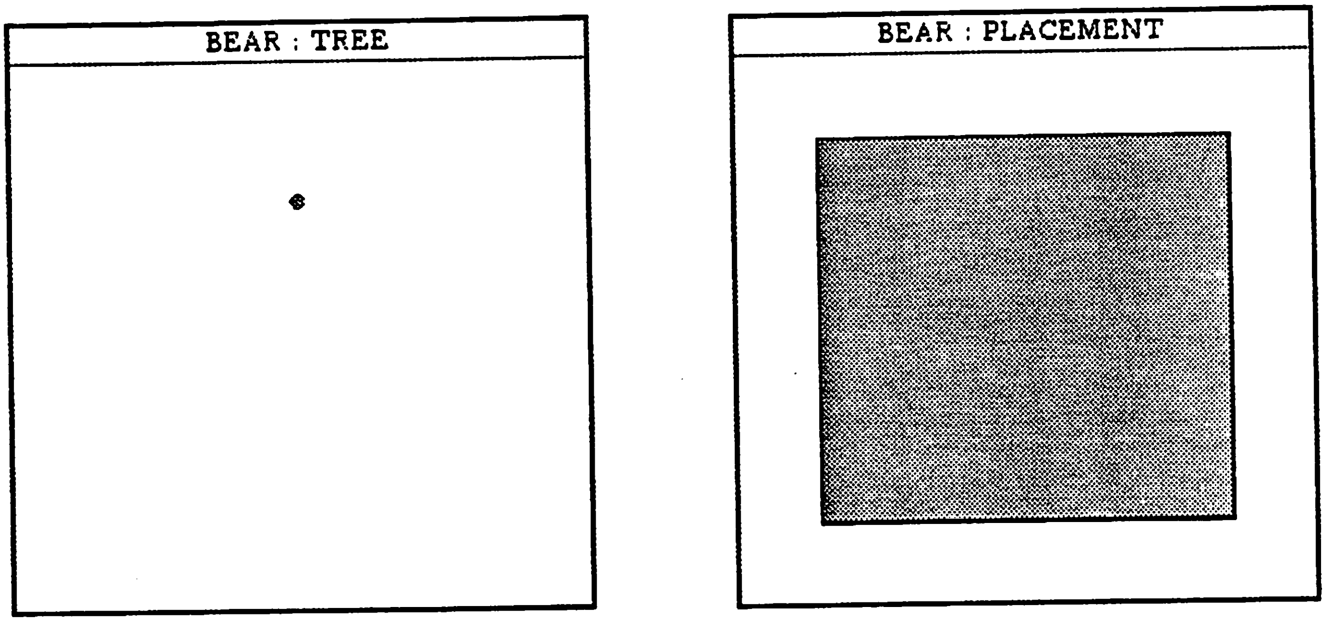


Fig. 5a.

Entering the series:

Next tree and floorplan level? **y**

Next tree and floorplan level? **y**

Next tree and floorplan level? **n**

will display the entire clustering tree and the placement of the example (see Fig. 5b) and returns the user back to the console.

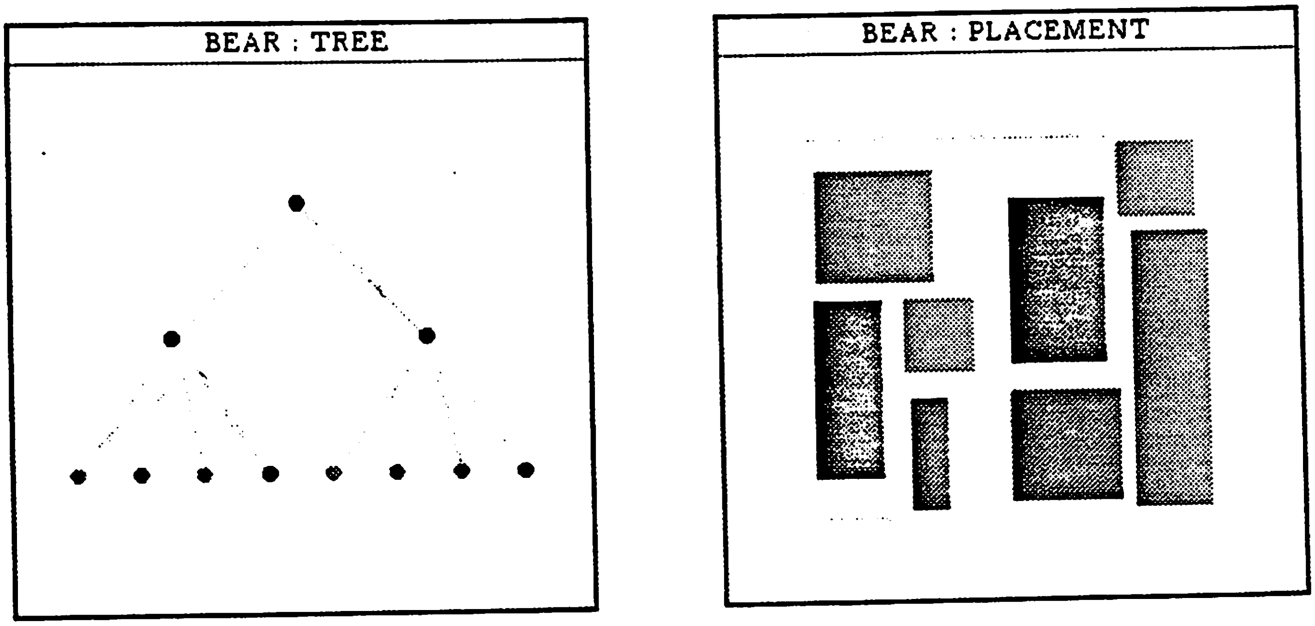


Fig. 5b.

7. Saving the placement.

If the placement is acceptable, then the user types:

bear > **wpl**

which stands for "write placement." The placement is then written to the chip window. (See Fig. 6.)

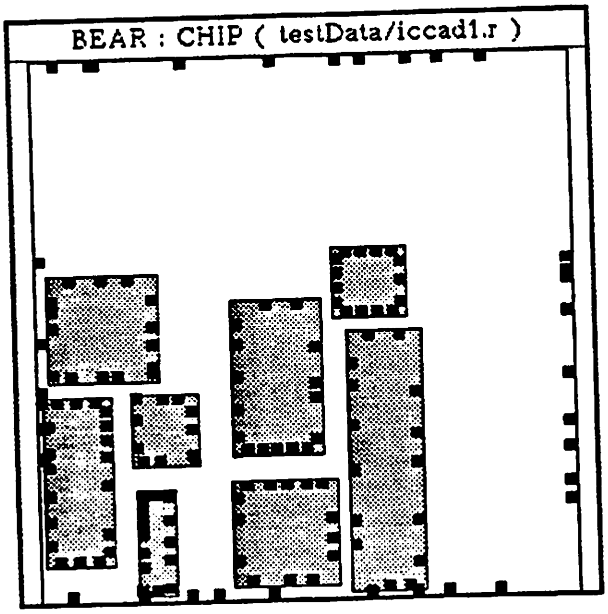


Fig. 6.

To close the tree and floorplan windows, the user should move the mouse over each window and type:

bear > **cw**

Alternatively, the user can type:

bear > **<control-d>**

over a window to accomplish the same task.

8. Global routing.

Now that the placement is completed, the example is ready for global routing:

bear > **gr**

The congestion factor and timing mode must then be specified (See section on Global Routing, III B, p. 32):

Congestion Factor [0]:

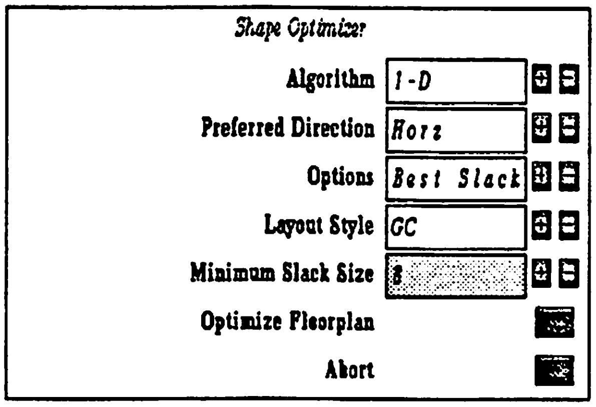
An IV window similar to the cluster IV window will appear showing the default values. When the global routing is done, the prompt will return.

9. Space optimization.

For further space minimization of the floorplan, the user may want to alter the shape of the cell blocks but still stay within the technology constraints. This task is accomplished by the shape optimizer command, but this time the user can try using the deck of cards menu system to enter in the command. The user should move the mouse over the chip window and hold down the middle mouse button. A stack of menus will immediatly emerge, each command sorted by action. The mouse should be moved to display the *Placement* menu. Choose the *shapeOptimize* entry. The corresponding keyboard command will be echoed to the screen:

bear > **so**

Another IV window will emerge. (See Fig. 7.)


Shape Optimizer

Algorithm	1-D
Preferred Direction	Horz
Options	Best Slack
Layout Style	GC
Minimum Slack Size	8

Fig. 7.

See the section on Shape Optimization (II E, p. 28) for explanation of variables. Clicking the mouse button next to *Optimize Placement* will begin the algorithm. When the algorithm is finished:

Shape Optimization finished

will be echoed, and the prompt will return to the console.

10. Global spacing.

Now the example will be ready for further global spacing. First, horizontal compaction will be invoked:

```
bear > hcm
Horizontal compaction is done.
```

Similarly, vertical compaction is invoked:

```
bear > vcm
Vertical compaction is done.
```

11. Detailed routing.

The next step is the detail routing:

```
bear > dch
```

This command stands for "define channel." The chip window will be redrawn showing the floorplan graph. (See Fig. 8.)

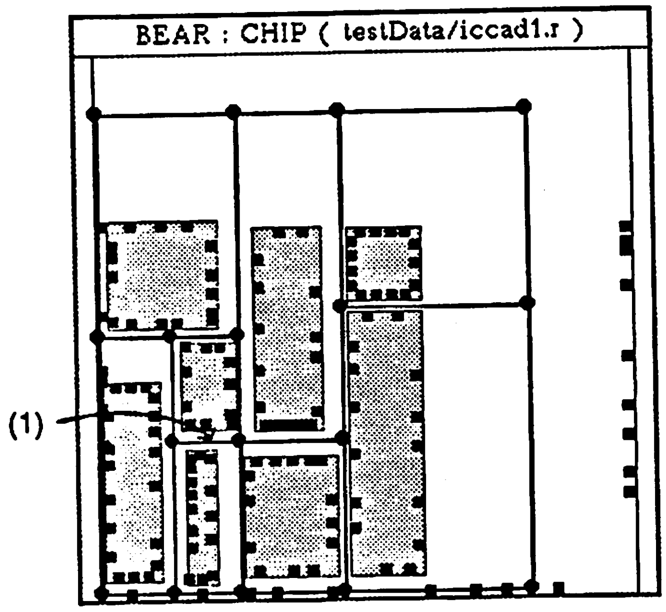


Fig. 8.

Independent channels will be highlighted. These channels are the valid channels that can currently be routed. A channel is picked by clicking the mouse over two junctions on the floorplan graph. At this point, this cursor will have changed from its normal cross to a circular cursor that is to be matched over a junction. The user will then be prompted:

Pick junction number one.

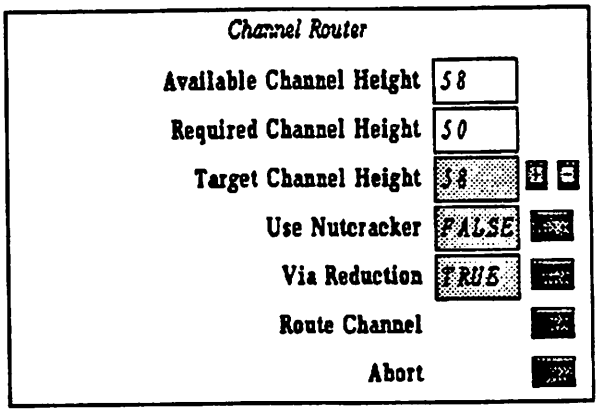
The user should then move the circle over an end junction of a highlighted channel and click the left mouse button. If the user did not select a valid end junction, the error:

Picking junction failed.

will be echoed to the console window, and the console prompt will return. Choose the channel marked (1) on Fig. 8. The second junction is specified in a similar manner.

Pick junction number two.

When this junction is successfully chosen, an IV window is displayed showing routing parameters. (See Fig. 9.)



Channel Router

Available Channel Height 58

Required Channel Height 50

Target Channel Height 58

Use Nutcracker FALSE

Via Reduction TRUE

Route Channel

Abort

Fig. 9.

Nutcracker is a local spacing routine which attempts to compact the routing to fit the available channel height. (See manual section on Detailed Routing.) All of the parameters are described in the Detailed Routing section of the manual, however, two particular parameters should be noted: available channel height and required channel height. If the required height is greater than the available height, then the channel should not be routed and the command should be aborted. Otherwise, there is enough space to accommodate the routing, and the user should click on *Route Channel.* Since sufficient space exists in this case, the channel can be routed.

When the routing is completed, the route cell will be drawn. The user can closely examine the route cell by defining a region to magnify:

bear > **ow**

Opening a window without any arguments allows the user to specify a region of a chip window to display. The user will be prompted to:

pick a rectangle

to define the region. The mouse button should be clicked and held down while moving the mouse itself to define the region. Once the button is released the window is created focused on that region. (See Fig. 10.)

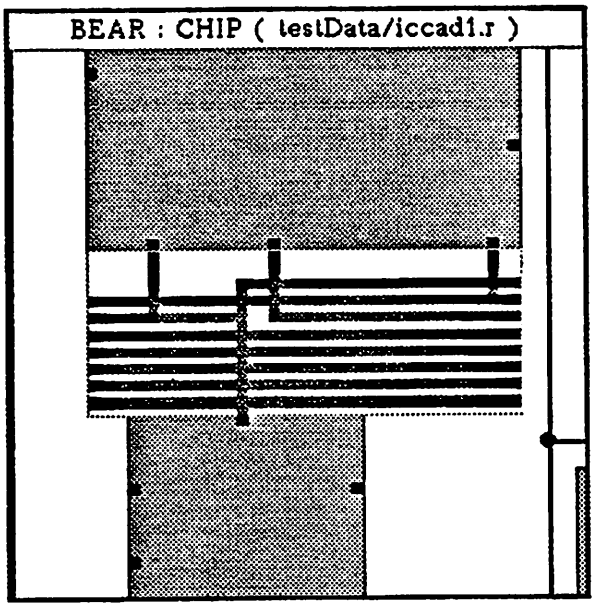


Fig. 10.

In any window except the console window, the user can zoom in, zoom out, magnify an area, or pan the view. In this case, moving the mouse to the newly created region and typing:

bear > **Z**

will show more detail of the routing. To destroy this window, type:

bear > **<control-d>**

Now the user can define another channel (2):

bear > **dch**

Pick junction number one.

Pick junction number two.

At this point if the user does not have enough space in the channel to fit the routing, the command must be aborted and decompaction must be done. If the channel to be routed is a vertical channel, horizontal decompaction is necessary. If the channel to be routed is a horizontal channel, vertical decompaction must be done as in this case:

bear > **vdcm**

Vertical decompaction is done.

The program is aware of the amount of space that the channel lacks for routing. The decompaction routines tries to adjust channel height accordingly so in the next attempt to define a channel:

bear > **dch**

Pick junction number one.

Pick junction number two.

the available height is as close to the required height as possible. If the channel is not on a critical path, however, the estimate may not be that close.

After routing a channel, it is always a good idea to run a decompaction routine to minimize the number of attempts to define a channel. In this case it is:

bear > **hdcm**

Horizontal decompaction is done.

The rest of the example is routed similarly. When all of the channels are routed, the floorplan will appear as in Fig. 11.

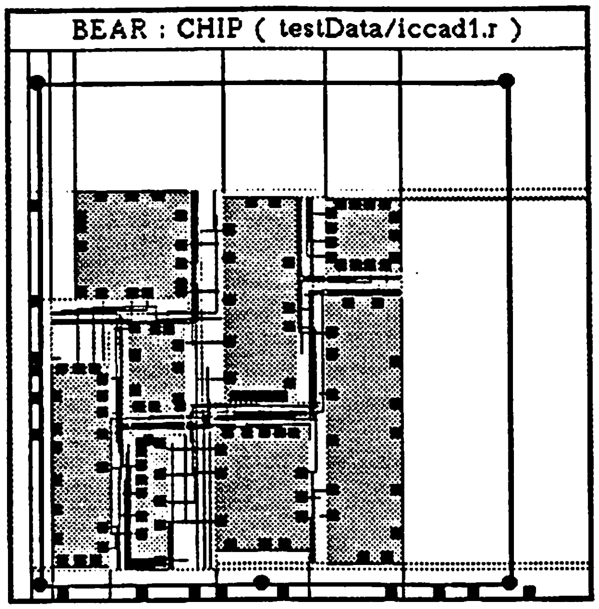


Fig. 11.

12. Resizing the parent cell.

Due to the compaction of cells, much space is left over in the parent cell, as can be seen in Fig. 11. This parent cell can be resized by:

bear > **rp**

At this point the mouse cursor changes to a circle and the user must select a corner or edge of the parent cell to drag to reduce the cell boundaries. In this case, the upper right corner of the cell is appropriate to begin dragging. As in the open window command, a rectangular outline is rubber-banded indicating the target size of the parent cell. When the resizing is finished, the console window will display:

Resizing...done.

(See Fig. 12.)

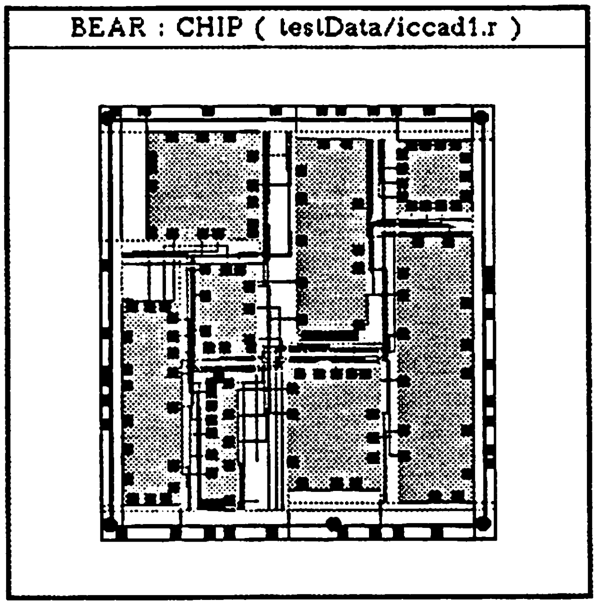

BEAR : CHIP (testData/iccad1.r)

Fig. 12.

If the cell blocks with routing are not centered in the parent cell, they can be moved all at once using the transform cell command:

bear > **tc -Im**

The **-Im** option specifies an interactive mode. When the circular cursor appears, the user must hold down the mouse button on any of the cell blocks and move the mouse. The outline of all the cells will be displayed giving the user an idea of where the cells will be oriented after releasing the button.

13. Ring routing.

The final step in the routing process is the ring routing:

bear > **rr**

An IV window will be opened displaying the default filename for the output of the ring router. When the ring router is finished, the layout is final. (See Fig. 13.)

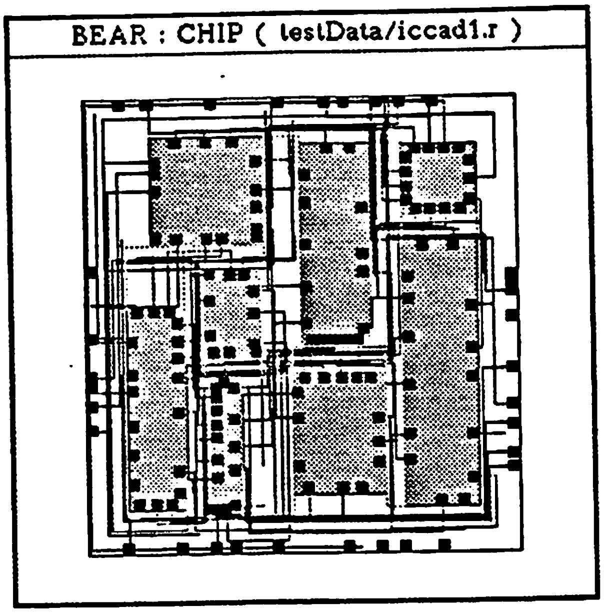


Fig. 13.

14. Saving the example.

The user may want to save the example in CIF format. The command is:

bear > **s -cif iccad1.cif**

where **iccad1.cif** is the name of the file to be written. A particularly useful feature of the save command is the scaling option. The user could have typed:

bear > **s -cif -scale 5 iccad1.cif**

All geometrical specifications in the CIF file will then be scaled by a factor of 5 in the horizontal and vertical directions.

15. Leaving the program.

To leave the program, the user must move the mouse to the console window and type:

bear > **cw**

One final dialog box is created prompting the user to confirm his or her request so that the layout is not accidentally erased.

II. Clustering, Placement, and Shape Optimization

The objective of the placement is to provide an arrangement of blocks which, after being routed, fits into an enclosing rectangle of minimum area with given height, width or aspect ratio. In order to achieve a high performance circuit, a concurrent goal is to minimize the length of connections. The BEAR placement algorithm combines the goal orientation of a top-down approach with the module orientation of bottom-up techniques. The result is a "meet in the middle" strategy. It considers the mutual dependency between placement and routing explicitly by incorporating a novel method of hierarchical routing area estimation. The placement process may be followed by a shape optimization phase which resizes individual modules so as to reduce the layout area. The user controls a set of input parameters to influence the clustering, placement or shape optimization stages. Due to the efficiency of the programs, it is possible to experiment with different parameter values to obtain more desirable results. However, the user may choose to use the default values which often give good results.

A. Clustering

The following description of the user interface of the BEAR placement program assumes that a chip window has been opened. In order to obtain a new placement, a hierarchical clustering tree must be built. Placement is then performed by traversing the tree top-down and placing the elements of each node optimally [4, 5]. The clustering algorithm can be invoked by typing **cl** while the cursor is in the chip window. A small pop-up window will ask the user to specify the type of clustering that is desired. Four algorithms are available:

- *Matching:* generates a clustering tree by optimal pairwise matching of modules and clusters [6].
- *Greedy:* does clustering based on a greedy heuristic.
- *Random:* randomly places modules in clusters.
- *Input from File:* reads the clustering tree directly from a file (see below). The best clustering results are often obtained by the matching algorithm.

After the desired algorithm is selected, a *Cluster Parameter* window will pop up. We shall describe each parameter, its effect on the clustering procedure, a typical range of values for it, and its default value. In particular, the matching algorithm parameter set is described. Parameters for other algorithms have similar meanings and ranges of values (see following table):

Clustering Parameter Set

parameter	*range*	*default*
Target Shape Flag	TRUE, FALSE	TRUE
Routing Area Flag	TRUE, FALSE	TRUE
Top-Down Adjustment Flag	0.1-2.0	1.0
Bottom-Up Adjustment Flag	0.1-2.0	1.0
Maximum Dimension Ratio	1.0-10.0	5.0
Maximum Area Ratio	1.0-10.0	2.5
Prompting	TRUE, FALSE	FALSE

1. Target shape generation flag.

If this boolean flag is set to *TRUE*, the clustering algorithm will generate the *target shapes* while building the clustering tree. Target shapes are the *optimal* shape goal of clusters derived by enumerating all possible topologies of elements of the lowest level clusters and then propagating this information recursively up the clustering tree. These shapes are then used as though they were the actual shapes of the non-leaf nodes in the subsequent top-down placement phase. Generating the target shapes is recommended since they often yield better placement topologies.

2. Routing area estimation flag.

If this flag is set to *TRUE*, the target shapes will be derived by allocating some routing area around the modules. In addition, in the placement phase that follows, at each node of the cluster tree some area is allocated for routing. This is recommended because it is often better to include the routing area *during* the placement phase rather than *after* the placement. If this flag is set to *FALSE*, the placement phase that follows will generate a *block packing*, and the routing area must be provided by interactive spacing of the modules.

3. Top-down and bottom-up routing adjustment factors.

After the input parameters are specified, the program starts to run, and at the end of each hierarchical level, information concerning the match between bottom-up and top-down routing area estimation is printed as standard output. If it turns out that the routing area allocated does not match the area needed, the routing area estimate can be increased with the *top-down routing adjustment factor* (value > 1.0) or decreased (value < 1.0) before a new clustering tree is generated. If the crude bottom-up estimation is wrong, it can be adjusted with the *bottom-up routing adjustment factor* in the same way. Although a mismatch does not cause any errors, it may produce strange results (especially if too much area is made available by the bottom-up estimation). The default value for both factors is 1.0.

4. *Maximum dimension and maximum area ratios.*

A clustering based only on connectivity information can result in a block shape mismatch that makes it impossible for the placement algorithm to avoid big *dead space* areas. It is our conjecture that two blocks do not match if (a) their areas and (b) the length of their longer sides are sufficiently different. A simple implementation is to prohibit the merging of block pairs whose areas or lengths differ by more than some ratio. The maximum dimension and maximum area ratios are computed based on the distribution of block sizes and areas available for clustering at each level of the hierarchy. Note that these parameters only appear in the dialog window for the matching and greedy clustering algorithms..

5. *Prompting flag.*

If this flag is set to *FALSE*, the clustering algorithm will use the internally computed values for the maximum dimension and maximum area ratios. If the flag is set to *TRUE*, it will stop and ask the user to enter new values (or accept the defaults) at each level of the hierarchy. The user may want to experiment with these ratios, however, the default values are often good.

B. Placement

After the clustering tree is constructed, the user can proceed with the placement. The placer can be invoked by typing **pl** while the cursor is in the chip window. To start the placer, a few input parameters are requested from the user (see following table):

Placement Parameter Set		
parameter	*range*	*default*
Chip Goal Shape	Fixed-X, Fixed-Y, Aspect-Ratio	Aspect-Ratio
Fixed-X or Fixed-Y	integer	computed
Aspect-Ratio	0.5-5	1.0
Area-Length Tradeoff	0-1.0	0.1
Lookahead	0-2	0
Prune	0-2.0	0.2

1. *Determination of the chip goal shape.*

The desired shape of the final layout can be specified in either of two ways: as goal aspect ratio (ratio of width to height) or as a fixed width or height of one dimension of the layout. The default shape in either case is a square. Although it cannot be guaranteed that the specified goal can be achieved exactly, the results are never far away from the desired value. Because the goal shape plays an important role in the computation of the objective function, it is often helpful to play around with this number to get the best result (for example, a change from 1.1 to 1.15 may have a big impact). This problem is alleviated if a 1-level lookahead (see below) with a reasonably large search region is specified.

2. Trade-off between area and sum of wire lengths.

On a scale from 0.0 to 1.0, the relative weight of the objective functions for area minimization and for wire length minimization can be influenced. 0.0 means that area minimization is most important; 1.0 emphasizes wire lengths. It is very difficult to know exactly what the optimal weighting for minimal wire lengths is because the dead space that is introduced by moving strongly connected blocks closer to each other may have a detrimental effect. The internal weight factors are adjusted so as to make it probable that a value of 0.1 gives the optimal solution. For some examples, however, it might be better to choose 0.05 or even 0.95.

3. Lookahead and pruning of the search tree.

To improve the placement results, the breadth-first traversal of the hierarchy can be complemented by a depth-first lookahead to improve the reliability of the objective function [4]. The user is free to specify different lookahead depths from different levels of the hierarchy and to narrow down the search space more or less drastically.

Because of the additional computational complexity of the lookahead, most of the time only 0 and 1 will be considered relevant. If a lookahead (> 0) is selected, the user is prompted for a constant that determines the width of the search. On every level the objective function of that level is computed. Only those possibilities with values of objective function lying between the minimal value and (1 + pruning constant) times the minimal value are explored.

For a lookahead of 0 (no lookahead) most time is spent in the last level to determine the orientations of the macrocells. To make the time spent on higher levels comparable to that time, a pruning constant of 0.5 is OK (but that depends very much on the example). A constant of 1.0 most of the time gives near-optimal results; only in few cases can the result be improved above values of 2.0.

C. Write Placement

The user can graphically examine the clustering tree and placement generated by typing **ow -tf** in the console window. The clustering tree and placement will be shown hierarchically, and the user should type **y** to see the next level. **q** or **<CR>** will terminate the command. In order to proceed with the global and detailed routing of the chip, it is necessary to transfer the placement from the internal data structure to the data base (and the chip window). This is accomplished by typing **wpl** in the console window (while the cursor is in the chip window).

D. File Interface for Clustering

To make it possible to keep the clustering tree constant after input changes, a clustering tree can be stored to and retrieved from a file. That file also offers the opportunity to edit the clustering tree. An example shows best how the file is organized:

```
 9
 1  18  23  24
 3   4   5  30
 6   7  20  27
 2  31  32  33
11  12   0   0
 8   9   1   0
13  17  28  29
14  15  16  19
21  22  25  26

 3
 1   9   0   0
 2   3   4   0
 5   6   7   8

 1
 1   2   3   0
```

A blank line starts a new clustering level, the lowest level appearing first. The second line gives the number of clusters on that level. Every cluster occupies one line with as many numbers as elements are allowed per cluster. If there are less elements than the maximal cluster size, zeros are used to pad the input lines to the standard length. The numbers on the first clustering level correspond to the sequence in which the blocks are stored in the input file. On the following clustering levels the numbers indicate the position in the preceding clustering level. For example on the second clustering level the first cluster consists of clusters 1 (blocks 1, 18, 23, 24) and 9 (blocks 21, 22, 25, 26) of the preceding level.

E. Shape Optimization

The user may have freedom in choosing the aspect ratio of a given module subject to some constraints. The assumption here is that although the functionality of the module is determined, the exact shape and pin locations of the module are not specified. Therefore, the placement algorithm may be followed by a global shape optimization which resizes and redistributes pins around the boundary of *flexible* modules in order to minimize an estimate of the layout area. If all modules are *stiff*, the user should proceed with the global and detailed routing without having to go through the shape optimizer.

After an initial placement of modules is derived, the *capacity* of the routing channels is known. However, in order to accurately estimate the layout area at each iteration, the shape optimizer must know the *density* of the channels. That is why *global routing* of the chip should be completed before the shape optimizer is called.

To run the shape optimizer, the user should type **so** in the console window. The user can specify a set of parameters to guide the shape optimizer. These parameters are as follows (see following table):

Shape Optimization Parameter Set

parameter	*range*	*default*
Design Style	GC, SC, GA	GC
Algorithm	1-D, 2-D	1-D
Preferred Direction	HORZ, VERT	HORZ
Slack Option	BEST-SLK, HALF-SLK, FULL-SLK	BEST-SLK
Minimum Slack Size	1-128	8

1. Design style.

This parameter specifies the type of modules on the chip. This is required since the shape optimizer must know which resizings are legal. In the current BEAR release only the *General Cell* (GC) design style (which says that the module dimensions may be continuously changed in either direction subject to aspect ratio constraints) is supported, and other styles (*Standard Cell*, (SC) and *Gate Array* (GA)) are not.

2. Algorithm.

This parameter specifies whether one-dimensional or two-dimensional shape optimization algorithms should be used. The 2-D algorithm performs simultaneous X- and Y- axis optimization. After the global routing both the block sizes and the estimated routing densities around the blocks are known. The longest or critical paths in either X- or Y- direction, which determine the extent of the layout, can thus be computed. The 2-D algorithm iteratively reduces the layout area by picking up a module with the largest resize capacity lying on a critical path (in either direction) and resizing it so that the module dimension along the critical path is reduced. The process terminates when no improvement in the layout area has been achieved after a certain number of previous iterations. For small macrocell circuits (less than 15-20 blocks), this algorithm is very efficient and gives excellent results. For larger circuits, because of the unpredictability of the changes made to the underlying topology, the algorithm may have a long run-time. Therefore, we allow for a two-pass 1-D shape optimization option (an X-direction pass followed by a Y-direction pass).

3. *Preferred direction.*

This parameter has no effect when the 2-D algorithm is selected. With the 1-D algorithm, however, it specifies the direction in which the chip dimension will be reduced. The other direction often remains unchanged. For large chips (> 15 modules), it is suggested that the 1-D algorithm be used because it keeps the circuit topology relatively unchanged and therefore is more likely to quickly converge to good solution.

4. *Slack option.*

This parameter determines the amount by which a given soft module is resized on each iteration of the algorithm. The horizontal slack of a module is the amount by which the X dimension of the module can be increased without increasing the chip X dimension. The vertical slack is defined similarly. Consider a block which lies on the longest path through the horizontal *space tile adjacency graph* [2, 3]. The *Best-Slack* resizes this block by an amount such that this block will at worst be placed on the second longest path through the vertical adjacency graph. The *Half-Slack* resizes the same block by half its slack in the vertical direction. The *Full-Slack* resizes this block by all the available slack in the vertical direction. The Full-Slack terminates much faster but often leads to poor optimization results. The program may even be trapped in a cycle whereby a block is resized in opposite directions alternatively until the phenomenon is detected and the program is automatically terminated. The Half-Slack takes more conservative resizing steps and is therefore slower. The Best-Slack has proven to be a good compromise between the speed of the Full-Slack approach and the quality of the Half-Slack approach .

5. *Minimum slack size.*

This parameter specifies the stopping parameter for the shape optimizer. Whenever the horizontal or vertical slack for each of the modules drops below the value of this parameter, the shape optimization terminates. Hence, the user may initially set the value of this parameter high (e.g. 32), and do the shape optimization. If the rough shape optimization result is acceptable, the user reruns the shape optimizer on the partially optimized chip, this time with a smaller value of Minimum Slack Size (e.g. 2). Before and after each shape optimization run, it is recommended that the user compact the chip so that the circuit topology becomes more representative of the final routed layout (see section on the global spacer, III C, p.35).

When the shape optimizer is run, information about the block being resized and repositioned at each iteration will be printed as standard output. The block being resized will be highlighted. At the end of the shape optimization phase the percentage of layout area reduction is printed to the standard output.

F. File Interface for Shape Optimization

In order to define the shape constraints for individual modules in a chip named "test.r", the user must create a file called "test.flex". The first line in the "test.flex" is the header line. It is of the form "NumFlexModules n", where *n* is the number of flexible modules that the user wants to read from the "test.flex". Each line following the header line has seven fields: "name minX maxY tarX tarY maxX minY". *name* is name of a flexible module, *minX, maxX* are the lower and the upper bounds on the horizontal dimension of the flexible module. *minY* and *maxY* are defined similarly. *tarX* and *tarY* are the target dimensions of the module, and in particular, are the dimensions assumed by the module in the initial placement phase. The parameters specify the legal range of aspect ratios for the module where *minAspect = minX / maxY* and a *maxAspect = maxX / minY* and the module may be resized to assume any aspect ratio between the two bounds. In particular, when *minAspect = maxAspect*, the module is considered stiff and is not resized. An example file follows:

```
                 NumFlexModules  6

B1   150    600   300   300   600   150
B2   150    576   180   480   360   240
B3   100    400   200   200   400   100
B4   220    480   440   240   660   180
B5   100   1480   200   740   400   370
B6   100    360   200   180   267   135
B7   150    200   300   100   400    75
B8   140    600   280   300   560   150
```

Notice that modules named "B7" and "B8" will be considered stiff by the shape optimizer.

III. The Routing System of BEAR

A. Overview of the Routing Process

Routing in a building block environment is a complicated task. Not only is the routing region irregular, but we also want to be able to move blocks during the routing process. The freedom to move blocks is a mixed blessing. It enables us to achieve more compact layouts than in the static case, but it complicates the problem tremendously. BEAR can handle block movement during the routing process and calls for a different routing data representation from that of conventional systems.

In the routing process, we assume that the placement of blocks has been predetermined. This placement is not completely rigid, but serves as a starting point. As we will see later, the initial placement can be deformed during routing, when the amount of distortion and the method of change depend on the user's actions. The starting placement can be arbitrary as long as blocks do not overlap, their sides are parallel to the x and y coordinates, and all cells are contained inside a chip area specified by the user.

After the placement has been read into BEAR, two *tile planes* are built. The horizontal tile plane consists of horizontal *tiles* [7]; the vertical tile plane consists of vertical tiles. Each of the tile planes has two kind of tile: *solid tiles,* which represent blocks; and *space tiles,* which cover the routing area. In the horizontal plane, the routing area is dissected horizontally into maximal horizontal stripes. In the vertical tile plane, the routing area is dissected vertically. The command *show tile property* can be used to view the tile planes. Invoking **stp -h** displays the horizontal tile plane; **stp -v** displays the vertical tile plane.

During the routing process, the *bottleneck* tiles play a key role. Intuitively speaking, these are tiles between the parallel edges of two neighboring modules, in the critical regions where congestion is most likely. Invoking **stp -b** displays the bottleneck tiles. For a more formal classification of tiles, please refer to [2].

The first step in the routing process is the determination of topologies and rough placement of all the nets. The nets' topologies and relative positions with respect to the blocks can be determined manually by using the *route net* command (invoked by **rn**), or can be automatically routed by using the *global route* command (invoked by **gr**). Nets routed manually by the user are treated as prerouted by the global router. The information about net routes is stored in the bottleneck tiles. The topology of a net may be viewed using the *show net property* command (invoked by **snp**).

Since it is very difficult to determine *a priori* how much space is needed to accommodate all the wires, block positions are adjusted after the global routing step. The adjustment is made by using the *compaction* (invoked by **cm**) or *decompaction* (invoked by **dcm**) commands. The user may wish to perform one-dimensional spacing (invoked by **hcm** for horizontal compaction, **vcm** for vertical compaction, **hdcm** for horizontal decompaction and **vdcm** for vertical decompaction) in order to move blocks and the global routes of nets so that the space between cells matches the estimate provided by the global router. Compaction removes extra space, while decompaction provides more space in congested areas of the chip.

Now we enter the detailed routing phase. The unrouted region is divided into straight channels and/or L-channels. First subregions are ordered, then one of those which can be routed at this time is selected. If we wish to select a *straight* channel to be routed now, the **dch** command is used. If we want to choose an L-channel, the **dlch** command is used. When **dch** or **dlch** commands are invoked, the legal channels are highlighted. When the detailed router completes its task, two previously disjoint chunks of layout are merged into one larger block. The nets exiting from the routed channel are fixed pins of the combined block. Such a combined block is called a *route block.* The results of partial detailed routing are now used to adjust the blocks' placement by performing the decompaction/compaction process again. After the adjustments, the remaining routing region is split and ordered again, the next channel is chosen, and the loop is executed until all the blocks are merged into one.

The last step is to connect this merged block to the pads on the periphery of the chip. This task is performed by the *ring route* command (invoked by **rr**).

Besides these basic routing steps, BEAR has the ability to calculate wire widths of power and ground nets by the *wire net* command (invoked by **wn**).

There are also many useful commands for showing the nets' topologies, checking connectivity etc., or creating or modifying examples.

B. The Global Router

1. Automatic global routing.

The global router is invoked by typing the **gr** command. The purpose of the global router is to determine the topologies and rough placement of wires on all unconnected nets. For each net, the global router determines through which bottleneck tiles the net will pass. This information is stored by means of *pseudo pins* which are attached to the open sides of bottleneck tiles. Each pseudo pin has an *internal* and an *external id.* Two pseudo pins connected inside a bottleneck tile have matching internal *ids*. Similarly, when the external *ids* of pseudo pins are the same, the pseudo pins are connected between different bottleneck tiles. Pins of cells, I/O pins and pins of route cells also have external and internal *ids*. Their internal *ids* are always 0. If a pin is inside a bottleneck, the external *id* matches the internal *ids* of corresponding pseudo pins; if a pin is outside a bottleneck, the external *id* matches the external *ids* of appropriate pseudo pins.

The global router used in BEAR takes a subregion in which routing has not yet been completed and determines a *cut* which separates it into two smaller subregions. When a net has pins or pseudo pins in both sides of the partition, the net crosses the cut line. For each such net, pseudo pins are inserted along the cut line in appropriate bottleneck tiles. This cutting process continues until each subregion on the list is free of bottlenecks. At each partitioning step, the linear assignment algorithm is used to determine where the net will be placed. The number of nets which can pass through a bottleneck tile is determined by the initial placement (geometrical dimensions of the tiles) and design rules (wire widths and spacings between them). The size of a bottleneck is measured by its *capacity*. The amount of available space inside bottleneck tiles can be decreased further by manually prerouting nets. The occupied space inside a bottleneck is measured by its *density*. The global router tries to place nets in bottleneck tiles so that the density (used space) does not exceed the capacity (available space) of each tile. If there is enough space for all the nets to cross the current cut line, then the assignment is performed. When there is not enough space, the global router increases the bottlenecks' capacities proportionally to their initial capacities until there is enough space for all the nets to pass through. Thus if the initial placement is too compact, the global router may produce many nets which do not take their shortest paths because it will try to utilize the existing area.

The global route command is controlled by a floating-point type parameter called the *congestion factor*. Its legal range is 0.0 to 1.0; its typical value is 0.0. This parameter controls the tradeoff between modifying placement and taking longer paths for nets. Small values will cause small modifications of the initial placement and, for placements with an underestimated routing area, may result in nets taking detours. Large values (close to 1) will result in nets taking shorter paths and more modified placement. A congestion factor larger than 0 artificially increases the capacities of the bottleneck tiles, but only for the purpose of the global router. The bottleneck tiles which are adjacent to the external bounding box are treated somewhat differently from the other bottleneck tiles: nets are assigned to pass through them only when necessary. This is because it is usually quite difficult to determine the dimensions of the bounding box, so even if it is overestimated, it will not cause nets to take detours through these regions.

The global router can take into account spatial constraints imposed on some nets. To execute this option the user has to set the *timing mode* to **true** in the dialog box, and then enter the name of a file which contains the following information:

number_of_nets [int],

vertical_layer_multiplication_factor [float],

net_name [string], max_net_length [float].

There must be as many *net_name, max_net_length* pairs as the value of *number_of_nets. Vertical_layer_multiplication_factor* is a parameter by which the lengths of vertical wires are multiplied in net length calculations; it is useful when wires on different layers have different conductances. If this distinction is not needed then the value of *vertical_layer_multiplication_factor* should be specified 1.0.

A detailed description of the global router can be found in [8].

Limitations: The global router requires the capacity of every bottleneck tile to be at least wide enough that one wire can pass through it. If a bottleneck tile does not fulfill this requirement a warning message is printed and the command aborts.

2. Manual global routing.

The user may wish to manually specify the global routes of some nets. The *route net* command provides methods for building the tree of a net. Nodes in the tree are terminals (pins) and nodes in the *floorplan graph.* Edges can be edges of the *floorplan graph,* pin to *floorplan graph* nodes, or pin to pin connections. *Route net* can be invoked by specifying the name of the net to be routed (**rn -n netname**) or by selecting a terminal on the screen. Thus, before executing the *auto global route* command, the user may preroute some nets. After selecting a net, the user is prompted with a menu of choices:

- Add a tree edge by selecting its two nodes. A valid tree edge is an edge from a pin of the net to a neighboring node of a *floorplan graph,* or an edge between adjacent *floorplan graph* nodes, or between two pins of the net if they are covered by a common edge of the *floorplan graph.* When an illegal edge is selected, a warning message is displayed and this edge is ignored.
- Delete tree edge (by selecting on the screen two nodes of the edge).
- Delete subtree (by picking any of its nodes).
- Draw a net — highlights the net being routed.
- Abort this command.

Limitations: A net must be only completely prerouted; it cannot be partially prerouted. If **rn** is used to route a net which will later have its wires sized (see paragraph 6), then no node of the net can have a degree exceeding 4.

3. Show commands which display results of global routing.

a. *Show net property* (invoked by **snp**.) The user is prompted to choose a net by selecting one of its pins. He may also specify a net by using **snp -n [netName]**. The specified net is highlighted on the screen. **snp -off** turns off the highlight.

b. *Show tile property.* **stp -h** displays horizontal tiles on the screen, **stp -v** displays vertical tiles, **stp -bp** displays both planes, **stp -b** displays bottleneck tiles, and **stp -co** displays cells only. When the **stp -lp** command is invoked with the mouse on a bottleneck tile in the window display, the list of pseudo pins attached to this bottleneck tile is printed on the X-window screen.

c. *Show pin property.* When the **spp** command is invoked, the user is prompted to choose a pin. The coordinate positions, type, external *ids,* etc., of the selected pin are printed.

C. Global Spacing

After the global router completes its job, the topologies and positions of all nets with respect to blocks are determined. Since it is quite difficult to estimate the routing area precisely, some bottleneck tiles will have more nets passing through them than their sizes permit. Similarly, some tiles will have less. The purpose of global spacing is to match the capacity of each bottleneck tile with its density as much as possible while preserving the existing nets' topologies.

There are two steps in global spacing. The first, global decompaction, is invoked by **dcm.** Its goal is to increase the size of the chip as little as possible so that negative mismatches (i.e. more nets than are allowed pass through a bottleneck) are eliminated. The second step, global compaction, is invoked by **cm.** Its goal is to reduce the size of the chip as much as possible without creating negative mismatches. The global spacer, working in either mode, selects a *ridge* which is a path through space tiles from one side of chip to the other. For horizontal compaction/decompaction the ridge goes from the top to the bottom of the chip; for vertical compaction/decompaction it goes from left to right. Ridges are chosen through bottlenecks with mismatches and then all objects on the top or the right side of the ridge are moved to increase or decrease the size of the ridge. For decompaction, ridges are selected from the smallest to the largest mismatch. For compaction, ridges are selected from the largest to the smallest mismatch. In addition, the ridges are selected alternately in the horizontal and vertical direction to preserve the topology of the placement.

The **cm** command compacts all mismatched ridges. **cm -l** allows the user to compact one ridge at a time.

The *push* command allows the user to manually select a ridge. First the user chooses a set of adjacent tiles from one side of chip to the other. Then he or she is prompted to give the amount that the ridge should be moved. The push command is invoked by **pu** for a horizontal ridge and **pu -v** for a vertical ridge.

Detailed description of the global spacing algorithm can be found in [3]. Each time blocks are moved, some bottleneck tiles may be destroyed and/or new ones created. Since we want to preserve the nets' topologies, after block movement the net connectivities are updated in the background. This process is invisible to the user and is not controlled by any external parameters. Details of the updating algorithm can be found in [2].

D. Iteractive Detailed Routing

1. Detailed routing iterative loop.

Detailed routing repeatedly executes the following steps:

a. The unrouted region is broken into straight and L-channels which are ordered appropriately. Please see [9] for details of the routing regions ordering algorithm. Horizontal channels can change their vertical dimensions, vertical channels can change their horizontal dimensions and L-channels can change both dimensions without affecting previously routed regions. The **t** command with option **o** displays the channels which can currently be routed.

b. A straight channel or an L-channel is selected from those currently feasible. The channel is defined by invoking the **dch** (straight channel), or the **dlch** (L-channel) command. These commands call detailed routers which perform routing but do not enter results into the data base. The available channel height (current size of channel) and required channel height are displayed on the screen. The user is prompted to either:

   - Route the channel (store the results in the data base). This option is used when the available height is not less than the required height and detailed compaction is not used.
   - Attempt to decrease the channel height using *detailed compaction* by checking the **Nutcracker** option on the screen and specifying the target height for the channel.
   - Abort the command. This option is used when the available channel height is less than the required height.

c. If the previous *channel route* command was aborted due to a mismatch between available and required space, then placement must be adjusted by the *local decompaction/compaction, manual move blocks* (invoked by **pu** or **pu -v**), or *transform cell* commands.

2. Detailed routers.

The channel router *Glitter* does the detailed routing. It is a gridless, variable width router. The **dch** command invokes *Glitter* on a straight channel; the **dlch** command invokes it on two straight subchannels created by dividing the L-channel. Details of the detailed routing algorithms can be found in [10] and [11].

When the **dch** command is invoked on the chosen channel, *Glitter* runs and displays the number of tracks it needs to connect all the wires of the selected channel. In addition to *Glitter's* results, the available height of the channel and the *target* height are displayed. Initially the target height is set equal to the available height. The user can compact routing produced by *Glitter* by decreasing the target height, setting the *Nutcracker* option to **true,** and checking **route.** This will invoke the channel compactor *Nutcracker. Nutcracker* will attempt to compact the initial detailed routing by inserting jogs. It will compact the channel as much as possible, but no more than the specified target height. After completing its job, *Nutcracker* displays its results and the user is prompted to either:

- Continue, if required and target heights match.
- Abort, if results are not satisfactory.

Glitter places horizontal wires on one layer and vertical wires on the other layer. This strategy may lead to many more vias than necessary. By default the detailed routing is followed by a *via reducer* which slides and removes unnecessary contacts. To turn this feature off, change the *via reduce* option from *true* to **false** in the box displayed by the **dch** command.

Detailed discussion of the algorithms used by the *channel compactor* and the *via reducer* can be found in [12] and [13], respectively.

Glitter routes L-channels and displays the results. At this point, results are not yet entered into the data base and the user is prompted to either:

- *Route channel* (store in the data base) if horizontal and vertical adjustments displayed are 0.
- *Abort* if router asks for horizontal or vertical adjustments.

By default, unnecessary vias produced by the detailed router are removed by the *via reducer.* To turn this feature off, change the *via reduce* option from *true* to **false. Nutcracker** cannot be invoked on an L-channel.

3. Placement adjustments.

This step is used to adjust a channel region to match the requirement specified by the detailed router. As we have seen in the section on detailed routers, if the detailed routing does not match the available area, the detailed routing command is aborted and the results are not entered into the data base. *Placement adjustment* is performed to correct this situation. There are two cases: either the available channel height was larger than required or it was smaller. In the first case, *local compaction* and in the second case, *local decompaction* may be used to modify the placement.

The local compaction step is used to compact the channel region to the number of tracks required by the previous detailed route. The orientation of the channel determines which compaction command is needed. A horizontal channel requires vertical compaction (invoked by **vcm**); a vertical channel requires horizontal compaction (invoked by **hcm**). The compaction command finds ridges across the chip and moves all blocks to the right or top of the ridge. If executed immediately after **dch,** the compactor will find the ridge which passes through the recently aborted channel. The amount the blocks are moved depends on the mismatch between the number of tracks needed by the router and the actual height of the channel. The compactor will not select ridges unless it results in a smaller chip area.

Local decompaction is used to add the number of tracks required to the channel that was previously detail routed and aborted. The orientation of the channel determines which decompaction command is needed. A horizontal channel requires vertical decompaction (the **vdcm** command); a vertical channel requires horizontal decompaction (the **hdcm** command). Local compaction and decompaction are based on similar principles.

The user may manually select a ridge for compaction or decompaction by invoking **pu** (horizontal ridge) or **pu -v** (vertical ridge).

Another method of adjusting placement is to manually move the blocks. Moving blocks maintains the global routing information and previously routed channels. To move a block manually, the *transform cell* command is used. First **tc** is invoked. The user is then prompted to choose a transformation: for manual cell moving, check the *move* option. Next, the user is prompted to choose the type of move: *delta x-y* or *interactive*. **dlch** gives x,y measurements for the delta x-y move. Finally the user is prompted to specify the cell on the screen.

E. Automatic Detailed Routing

The automatic router is invoked by typing the **ar** command. This command routes all channels in the chip, decompacting when necessary. The command can be invoked with the **-i** option to allow the user to route one channel at a time.

When **ar** is invoked, a dialog box is created with options for applying *Nutcracker* and *via reducer* to all channels in the chip.

F. Ring Route

The last step in detailed routing is ring route (invoked by **rr**), which connects the core of a chip to the I/O pads at the periphery. The ring router expects all signal wires to be the same width. Wires specified as power/ground can be of arbitrary widths.

Limitations: Ring route cannot handle power pads at the corners of the bounding box or vertical constraints between power nets.

G. Wire Widths Sizing

BEAR is capable of determining the widths of power and ground nets so that the area of wire segments is minimized while fulfilling electromigration and voltage drop constraints. Details of the wire sizing algorithm can be found in [14].

The *wire net* command is invoked by typing **wn**. **wn** expects that a net whose wire widths are to be determined has a certain structure, called *gtTree*, already built in the data base. If the user's intention is to calculate the wire widths of a net which was manually global routed, then *gtTree* already exists and no additional action is necessary. If the user wishes to determine the wire widths of a net which was automatically global routed, then a *gtTree* must be built using the *build tree* command (invoked by **bt**). **wn** requires a *wire technology* file. In this file, which can have an arbitrary name, the parameters needed by *wire net* are stored. These parameters are as follows:

> *grid [float]:* specifies how many microns correspond to one grid line;
>
> *conductance [float]* (in A/V): specifies conductance of wires;
>
> *curPerMicron [float]* (in A/micron): electromigration constant (i.e. the max branch current ≤ curPerMicron * width);
>
> *minWidth [int]* (in grid lines): specifies the minimum acceptable wire width;
>
> *timeSteps [int]:* specifies in how many time steps the calculations are to be performed (usually 1);
>
> *flag [int]:* used to set appropriate parameters for **wn.**

There are three parameters which need to be set in *wire net: elect_flag, volt_flag* and *feasible_flag*. When *elect_flag* is 1, electromigration constraints are included in calculations; if it is 0 then they are not. When *volt_flag* is set to 1, voltage constraints are taken into account; if it is set to 0 then they are not. When *feasible_flag* is set to 1, only a feasible solution is sought; if it is 0 then an optimal solution is calculated.

These parameters are calculated as results of the following bitwise operations:

elect_flag = PGR_ELECT_FLAG & *flag*
volt_flag = PGR_VOLT_FLAG & *flag*
feasible_flag = PGR_FEASIBLE_FLAG & *flag*

where PGR_ELECT_FLAG = 01, PGR_VOLT_FLAG = 02, PGR_FEASIBLE_FLAG = 04. Thus for example, *flag* set to 05 causes elect_flag = 1, volt_flag = 0 and feasible_flag = 1.

When the *wn -n [netName]* command is invoked, the user is prompted to give the following information:

- Specify feasible voltage drops (in V) and current requirements (in A) for receiving terminals.
- Enter the name of the *wire* technology file.
- Specify which terminals of the net (by selecting them on the screen) are current sources.
- Invoke wire width calculations — can be executed after all above information has been specified.

Appendix 1. Input Format Specifications

BEAR input data are entered from two files. The first file contains block dimensions and pin-net specifications. It is called a *routing file* and its name must be *something.r*. The second file contains descriptions of design rules and is called a *technology file*. Its name must be *something.tech*. The first portion (up to the dot) of both files must be the same.

A. Input Format for a Routing File

1. The input text file format.

```
SN<number of nets>
{ top level module data }
$
{ module data at this level }
         .       .
         .       .
         .       .
$
```

2. The format of module data.

```
MOD                 /* top level module */
<x> <y>             /* integer origin coordinates, all module coordinates
                       are relative to this position */
<module name>       /* up to 8 characters */
<module flag>       /* 1 = top routing module; 0 = bottom module */
<x1> <y1>           /* corner coordinates of the module in the
                       counterclockwise direction */
<x2> <y2>
 . .
 . .
 . .
$
T                   /* terminals */
<x> <y> <name> <direction> <type> [ <layer> <width1> <width2> <p/g flag>
<current> <voltage> ]

/* <x> <y>:      terminal coordinates relative to the origin of the module */
/* <name>:       the name of the net that the terminal belongs to, up to 40
                 characters */
/* <direction>:  routing direction (0: west, 1: south, 2: east, 3: north) */
/* <type>:       terminal type (0: floating, 1: edge fixed, 2: fixed) */

[ Optional specifications ]

/* <layer>:      terminal layer (1: layer 1, 2: layer 2, 12: layer 1 or 2) */
/* <width1>:     width on layer 1 */
/* <width2>:     width on layer 2 */
/* <p/g flag>:   power/ground flag (0: signal, 1: ground, 2: power) */
/* <current>:    current requirement of the p/g terminal */
/* <voltage>:    voltage drop of the p/g terminal */
   .     .
   .     .
   .     .
$
```

B. Input Format for a Technology File

```
                                                               /* number of layers */
N

w(1)  mxi(1)   sr(1)   plc(1)   prc(1)                         /* Layer Rule */
w(2)  mxi(2)   sr(2)   plc(2)   prc(2)
    ..
    ..
    ..
w(N)  mxi(N)   sr(N)   plc(N)   prc(N)

sz(1)   i(1)     r(1)    c(1)    prc(1)                        /* Hole Rule */
sz(2)   i(2)     r(2)    c(2)    prc(2)
    ..
    ..
    ..
sz(M)   i(M)     r(M)    c(M)    prc(M)


m(1,1)   p(1,1)   m(1,2)   p(1,2) ... m(1,N)   p(1,N)    /* LayerLayer Rule */
m(2,1)   p(2,1)   m(2,2)   p(2,2) ... m(2,N)   p(2,N)
    ..
    ..
    ..
m(N,1)   p(N,1)   m(N,2)   p(N,2) ... m(N,N)   p(N,N)


s(1,1)   s(1,2)   ......   s(1,M)                          /* HoleHole Rule */
s(2,1)   s(2,2)   ......   s(2,M)
    ..
    ..
    ..
s(M,1)   s(M,2)   ......   s(M,M)


m(1,1)  ov(1,1)   m(1,2)  ov(1,2) ... m(1,M)  ov(1,M)  /* LayerHole Rule */
m(2,1)  ov(2,1)   m(2,2)  ov(2,2) ... m(2,M)  ov(2,M)
    ..
    ..
    ..
m(N,1)  ov(N,1)   m(N,2)  ov(N,2) ... m(N,M)  ov(N,M)


NOTE:  M = number of Hole types = N -1
```

The following explains the table above.

1. Layer Rule: one line for each layer (see Fig. 14).

For layer j:

w(j)	=	minimum wire width
mxi(j)	=	maximum current carrying capacity per micron wire width
sr(j)	=	sheet resistivity per square micron
plc(j)	=	plate capacitance per square micron
prc(j)	=	capacitance per micron perimeter length (fringing capacitance)

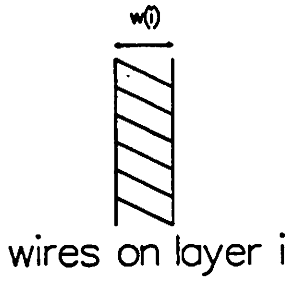


Fig. 14.

2. Hole Rule: one line for each type of hole (see Fig. 15).

For hole j:

sz(j)	=	size
i(j)	=	current
r(j)	=	resistance
c(j)	=	capacitance

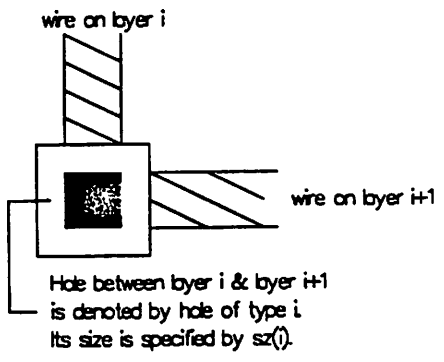

wire on layer i
wire on layer i+1
Hole between layer i & layer i+1
is denoted by hole of type i.
Its size is specified by sz(i).

Fig. 15.

3. LayerLayer Rule: one line for each layer (see Fig. 16).

For layers i and j:

m(i,j)	=	minimum spacing between wires between layer i & layer j
p(i,j)	=	maximum (longest) parallel wiring between layer i & layer j

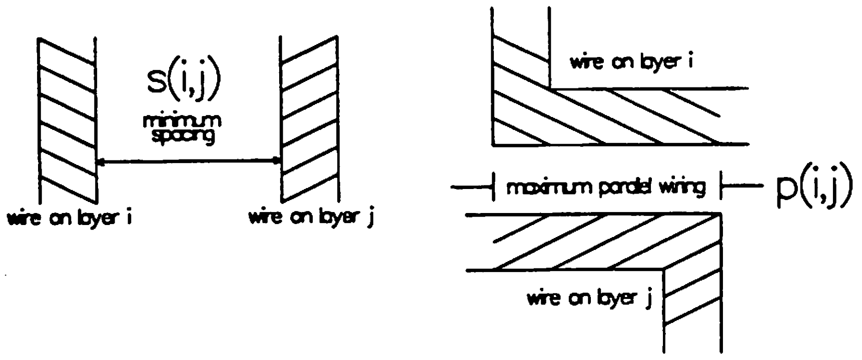


Fig. 16.

4. HoleHole Rule: one line for each type of hole (see Fig. 17).

s(i,j)	=	minimum spacing between hole i and hole j

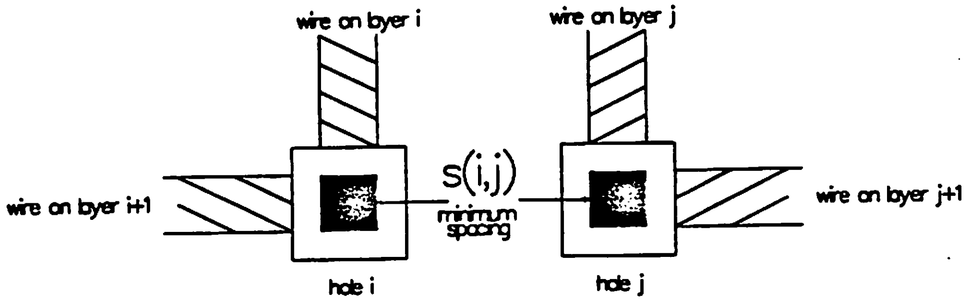


Fig. 17.

5. LayerHole Rule: one line for layer (see Fig. 18).

m(i,j)	=	minimum spacing between layer i & hole j
ov(i,j)	=	minimum overlap width between layer i & hole j

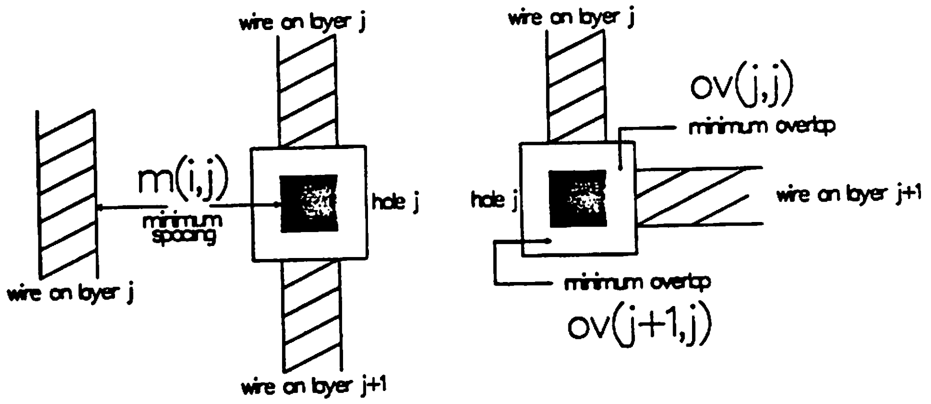


Fig. 18.

Appendix 2. X Defaults

BEAR allows you to preset defaults in a customization file in your home directory called

~/.Xdefaults

The format of the file is *programname.keyword:string*. *BEAR* obeys the convention for 'MakeWindow' defaults. Keywords recognized by *BEAR* are listed below.

BlackAndWhite	If **on**, a black and white color scheme will be used even on a color display so that programs that dump windows to printers will work.
ReverseVideo	If **on**, reverse the definition of foreground and background colors on black and white displays.
Background	Determines the background color for all windows other than the console window.
Border	Determines the border color for all windows other than the console window.
BorderWidth	Determines the border width for all windows other than the console window.
Foreground	Determines the foreground color for all windows other than the console window.
Font	Determines the font for text in all windows other than the console window.
Highlight	Determines the highlight color for all windows other than the console window.
Mouse	Determines the mouse cursor color for all windows.
Text	Determines the color of prose printed in a window.
Console.Background	Determines the background color of the console window.
Console.Border	Determines the border color of the console window.
Console.Cursor	Determines the cursor's color in the console window.
Console.Geometry	Geometry specification for the placement of the console window on start up.
Console.BoldFont	Determines the console window's boldface font which will be used to show everything the user types.
Console.ItalicFont	Determines the console window's italic font which will be used to show error messages.
Console.NormalFont	Determines the console window's normal font which will be used to show basic system messages.

Console.BoldColor	Determines the color of the boldface font within the console window.
Console.ItalicColor	Determines the color of the italic font within the console window.
Console.NormalColor	Determines the color of the normal font within the console window.
Chip.Cell	Determines the color of cells on the chip.
Chip.RouteCell	Determines the color of route cells on the chip.
Chip.DummyCell	Determines the color of dummy cells on the chip.
Chip.CellBorder	Determines the border color of all cells on the chip, the pin color, as well as the color of the floor plan graph.
Chip.Pin	Determines the color of the pins for the chip.
Chip.Background	Determines the background color of the chip.
Chip.hchannel	Determines the horizontal channel color.
Chip.vchannel	Determines the vertical channel color.
Chip.lchannel	Determines the L-shaped channel color.
Chip.Net1	Determines the color of net one on the chip.
Chip.Net2	Determines the color of net two on the chip.
Chip.Net3	Determines the color of net three on the chip.
Chip.HorzBottleNeckTile	Determines the color of horizontal bottleneck tiles.
Chip.VertBottleNeckTile	Determines the color of vertical bottleneck tiles.
Chip.HorzDominantTile	Determines the color of horizontal dominant tiles.
Chip.VertDominantTile	Determines the color of vertical dominant tiles.
TF.leaf1	Determines the color of leaf number one in the tree and floorplan windows.
TF.leaf2	Determines the color of leaf number two in the tree and floorplan windows.
TF.leaf3	Determines the color of leaf number three in the tree and floorplan windows.
TF.leaf4	Determines the color of leaf number four in the tree and floorplan windows.
TF.leaf5	Determines the color of leaf number five in the tree and floorplan windows.
TF.node	Determines the color of the nodes in the tree windows.
TF.edge	Determines the color of the edges in the tree windows.
Cif.BND0	Determines the color of cif boundary zero.
Cif.BND1	Determines the color of cif boundary one.
Cif.BND2	Determines the color of cif boundary two.

Cif.NC	Determines the color of cif nMos contact cut color.
Cif.NM	Determines the color of cif nMos metal color.
Cif.NP	Determines the color of cif nMos polysilicon color.
Cif.TRM	Determines the color of cif text.

The following defaults are for the IV windows:

Iv.Background	Set the background color. Default is light grey on color displays, black on monochrome.
Iv.BorderColor	Set the border color. Default is black on color displays, white on monochrome.
Iv.BorderWidth	Set the border width of the main IV window, and the border around the edit region windows. Default is 1.
Iv.ButtonColor	Set the color of the buttons. Default is yellow on color displays, white on monochrome. For best results, choose a non-dark color.
Iv.CursorColor	Set the color of the mouse cursor. Default is green on color displays, white for monochrome.
Iv.EditBackground	Set the background color of the edit region. Default is light blue on color displays, black for monochrome.
Iv.EditFont	Specify the font to print the edit region. Default is 6x10.
Iv.EditFontColor	Set the font color of the edit region. Default is red for color displays, white for monochrome.
Iv.EraseValue	If **on** clear the edit region upon editing the variable. The default is **off**. Note that data can still be recovered by CONTROL_U.
Iv.Padding	Specifies the extra padding above and below each IV row (text and variable). The default is 2.
Iv.TextFont	Specify the font to print the documentation field. Default is 6x10.
Iv.TextFontColor	Set the font color of the documentation field. Default is blue for color displays, white for monochrome.
Iv.TitleFont	Specify the font to print the title. Default is 9x15.
Iv.TitleFontColor	Set the font color of the title Default is dark slate blue for color displays, white for monochrome.

Appendix 3. OCT Interface

The following is a description of how to read from and write to OCT in *BEAR*.

First, *BEAR* must be executed before the read-from or write-to OCT commands can be performed. Once *BEAR* is running, to load a chip from the OCT database, issue the following open window command:

ow -oct cellname viewname [output_cellname [output_viewname]]

where **cellname** is the name of the OCT cell to be read, **viewname** is the name of the OCT cell's view, and **output_cellname** is the name of the OCT cell used temporarily in the reading process.

This OCT cell is an exact copy of the OCT cell described by **cellname** and **viewname**, but with two additional bags to facilitate in reading and writing back. Also, **output_cellname** is the name of the OCT cell to be written back out when the *save* command without any optional information, **s -oct**, is used later. (The default for the optional **output_cellname** is *macout*.) **output_viewname** is the name of the OCT cell's view which is temporarily used and is to be written back out. (The default for the optional **output_viewname** is the same name as the **viewname.)**

When writing to OCT in *BEAR*, issue the following save command,

s -oct [[-r] output_cellname output_viewname]

where the **-r** option is for saving routing information to OCT and **output_cellname** is the name of the OCT cell to be written out to OCT. This name is only optional when saving an OCT cell that has been read in by the *open window* command described above, has never been saved after being read in, and has not been changed other than the placement of its cells. The default for the optional **output_cellname** is the same name as the **output_cellname** used in the *open window* command (above) when reading in from OCT. **output_viewname** is the name of the OCT cell's view to be written out and is also only optional under the same conditions as the ones described just above in **output_cellname** of this command. The default for the optional **output_viewname** is the same name as the **output_viewname** used in the *open window* command (above) when reading in from OCT.

The *save* (write back) command,

s -oct

is a much faster write mechanism than the *full save* (write) command with options. It uses knowledge from the *open window* (read) command and only updates the placement of the cells.

Example 1: Make placement modifications on an OCT chip and save in macout.

(Open window commands are typed with the cursor in the *BEAR* console window, and all other commands, *save* and *close window*, are typed with the cursor in the window displaying the cell read in from OCT using *open window*.)

```
ow -oct foocell fooview
```

(Opens a window displaying the information read in from OCT cell, foocell, and view, fooview.

[placement modifications]

.

.

```
s -oct
```

(Saves the placement information in the window by writing the modifications to OCT cell default, macout, and view default, fooview.)

```
cw
```

(Closes the window.)

Example 2: Make placement modifications on temporary file to be saved in new cell foocell.

```
ow -oct macout fooview foocell1 fooview1
```

(Opens a new window containing the previously saved OCT cell, *macout*, and view, *fooview*.)

[more placement modifications]

.

.

```
s -oct
```

(Saves the placement information in the window by writing the modifications to OCT cell, *foocell1*, and view, *fooview1*.)

.

.

Example 3: Save placement and routing modifications.

[more placement modifications, routing, and/or creating, deleting, modifying new cells, pins, and nets]

.

.

s -oct -r foocell2 fooview2

(Saves the placement and routing information in the window by creating and writing to OCT cell, *foocell2*, and view, *fooview2*.)

cw

(Closes the window.)

Example 4: Read in BBL file and save it in OCT database.

ow -bbl foobbl

(Opens a new window displaying the information read in from BBL file, foobbl)

.

.

[placement modifications and/or creating, deleting, modifying new cells, pins, and nets]

.

.

s -oct foocell3 fooview3

(Saves the placement information in the window by creating and writing to OCT cell, *foocell3*, and view, *fooview3*.)

cw

(Closes the window.)

Unless otherwise specified during the *open window* command with the OCT option, the default output OCT cell name is *macout* and the view name is the same as the view name read in. Any type of modifications can be done on the data that has been just read into *BEAR* from OCT with the *open window* command. After all the necessary changes have been made on the data, *save* command (**s -oct**) should be issued to write back only the placement data to the default *OCT cell* and *view.* This *save* command can be used to write back placement data only once after every read from OCT. Thus, after a write back, the window must be closed and a new window must be opened to read in the changed data for any new modifications made thereafter to be saved properly. Alternatively, the *full save* command with all the options can be issued. This action will write the placement and routing information to the newly created OCT cell and view named in the options. If only placement data is to be written out to a new *OCT cell* and *view* then the full *save* command with all the options minus the **-r** option should be issued. Other types of data such as *BBL read into BEAR* by the *open window* command or new information created inside *BEAR* can be written to OCT by using the full *save* command with the options.

Appendix 4. XDM: X Dialog Manager [*]

A. General Information

1. *Synopsis.*

#include <X/Xlib.h>
#include "XDM.h"

2. *Modification and control routines.*

int XDMInit(prName) char *prName;

int XDMModify(dialog, fieldId, name, value, ..., XDM_END) Window dialog; int fieldId;

int XDMQuery(dialog, fieldId, name, value, ..., XDM_END) Window dialog; int fieldId;

int XDMDelete(dialog, fieldId) Window dialog; int fieldId;

int XDMPost(dialog, x, y, func, options) Window dialog; int x, y; int (*func)(); int options;

int XDMFilter(event, dialog, fieldId) XEvent *event; Window *dialog; int *fieldId;

int XDMEnd(dialog) Window dialog;

3. *Field creation routines.*

Window XDMDialogCreate(parent) Window parent;

int XDMTextCreate(dialog, id) Window dialog; int id;

int XDMButtonCreate(dialog, id) Window dialog; int id;

int XDMBlendCreate(dialog, id, relId) Window dialog; int id; int relId;

int XDMEdRegCreate(dialog, id) Window dialog; int id;

int XDMRowColCreate(dialog, id) Window dialog; int id;

int XDMForeignCreate(dialog, id, w, h, bdrSize, bdr, bgnd, minSize, optSize, realSize, thePos, delFunc, win) Window dialog; int id; int w, h; int bdrSize; Pixmap bdr, bgnd; int (*minSize)(); int (*optSize)(); int (*realSize)(); int (*thePos)(); int (*delFunc)(); Window *win;

4. *General purpose and error routines.*

int XDMForEach(dialog, func, arg) Window dialog; int (*func)(); XDMPointer arg;

[*] Program written by David Harrison, University of California, Berkeley.

Window XDMFindDialog(field) Window field;

int XDMTypeQuery(dialog, fieldId) Window dialog; int fieldId;

char *XDMError()

5. Overview.

XDM is an interactive forms-based input system for X. It provides means for displaying and controlling a window which may contain text, buttons, blender controls, type-in fields, and foreign windows. These dialogs can be used to ask user for input in a easy to use, aesthetically pleasing manner.

XDM is intended for use in conjunction with Xlib, the C Language X Window System Interface Library. The programmer builds a dialogs using field creation routines, posts them using *XDMPost,* and then routes all events in the program's main event loop through the dialog event handler, *XDMFilter.* The event handler handles all events associated with the dialog and ignores all other events. Furthermore, if some action on the part of the user requires some response from the application, *XDMFilter* indicates this and returns the appropriate information.

Although XDM is written in standard C, the programming style is object oriented. Once created, the basic components of XDM respond to a set of pre-defined messages which are passed to a component using the *XDMModify* and *XDMQuery* routines. Any changes to the components are reflected immediately in the corresponding dialog window.

XDM components are created in a hierarchical fashion. At the top level, there is the *dialog* component. A dialog is a window which may contain other components. Some of these components may themselves contain other components, forming a tree. This tree is used to define the *control path* for input events to a dialog. If an input event is not handled by a lower level component, it is passed up automatically to its parent. Its parent may handle the event, allow it to be transferred up to its own parent, or pass the event down to one of its children.

For example, a button component may contain one other component (normally text) which is considered "inside" the button. Normally, the component inside a button ignores mouse button events. These events are passed up to the button to be handled. The button may then send messages down to the text component indicating it should reverse its colors.

Components are identified by the dialog that contains them (an X window), and a *fieldId.* The fieldId is a positive integer returned by all object creation routines. This identifier is assigned by the containing dialog component in increasing order starting from zero. Thus, as long as the components are created in the same order, they will always have the same fieldId.

New components are created by routines which are specific to the component. However, all of these routines require the programmer to identify the Window of the containing dialog, and the fieldId of the parent component (or XDM_WINDOW if it is a direct child of the dialog). All of these routines guarantee to return a fieldId for the newly created component.

Once a component has been created, messages can be sent to it by the generic routines **XDMModify, XDMQuery,** and **XDMDelete. XDMModify** and **XDMQuery** are the primary means of changing and querying the state of all components. Since components may have many different options, these routines have *variable length argument lists.* The first two parameters are always the dialog and fieldIds which uniquely specify a particular component. The dialog itself can be modified by specifying XDM_WINDOW as the fieldId. The remaining arguments are name/value pairs terminated by the end-of-list identifier XDM_END. The order of these name/value pairs is not significant. The message names are listed in XDM.h and in the component descriptions below. The component itself defines what name/value pairs are legal for that type of component. However, for the use of the dialog itself and any formatting components, all objects support messages for setting and querying their size and position.

6. Example.

A program using the facilities of XDM is structured as an event handling loop. The library is initialized by the routine **XDMInit.** This routine reads the users *~/.Xdefaults* file (see the defaults section near the end of the man page for details) and initializes the package. New dialogs and components are then created using the various component creation routines described in the sections that follow. Dialogs are placed on the screen by the **XDMPost** routine. After posting one or more dialogs, the user enters an event loop where X events are filtered through the routine **XDMFilter. After the user interaction with a dialog is complete, the programmer can use XDMEnd** to delete dialogs.

The following programming example shows how these routines interact:

```
/*
 * A program to display one button with the text
 * "Hello, World" inside it.  When the user clicks
 * on the button, the dialog finishes.
 */

#include <X/Xlib.h>
#include "XDM.h"

main(argc, argv)
int argc;
char *argv[];
{
    Display *theDisp;
    Window TheDialog, ReturnDialog;
    XEvent theEvent;
    int ButtonField, TextField;
    int ReturnField;
theDisp = XOpenDisplay(argv[1]);
  XDMInit(argv[0]);
  TheDialog = XDMDialogCreate(RootWindow);

  /* XDM_WINDOW is used to indicate a direct child of the dialog */
  /* (Note: ButtonField will be zero after the call) */
  ButtonField = XDMButtonCreate(TheDialog, XDM_WINDOW);

  /* Tell the button to notify us when it is pressed */
  XDMModify(TheDialog, ButtonField, XDM_SETSIGNAL, 1, XDM_END);

  /* Create a text component which is a child of the button */
  /* (Note: TextField will be one after the call) */
  TextField = XDMTextCreate(TheDialog, ButtonField);

  /* Modify the value of the text field */
  XDMModify(TheDialog, TextField, XDM_TEXT, "Hello, World", XDM_END);

  /* Display the dialog */
  XDMPost(TheDialog, 100, 100, 0, 0);

  /* Event handling loop */
  for (;;) {
     XNextEvent(&theEvent);

     if (XDMFilter(&theEvent, &ReturnDialog, &ReturnField) < 0) {
        /* Signal event: ReturnDialog and ReturnField are set */
        if (ReturnField == ButtonField) {
           /* Obviously true in this case */
           XDMEnd(TheDialog);
           exit(0);
        }
     }
  }
}
```

7. General control routines.

XDMInit Initializes the XDM Package and reads the user's *~/.Xdefaults* file. *prName* is the name of the program (normally argv[0]). The defaults for the package are listed in the defaults section near the end of this manual.

XDMModify This routine modifies the specified attributes of the component in *dialog* with unique id *fieldId.* If *fieldId* is XDM_WINDOW, *dialog* is interpreted as the window of the component itself. This is how dialogs themselves are modified. The legal names to use in the variable length name/value pairs are listed in XDM.h and in the sections that follow. The last argument to the routine should always be XDM_END.

XDMQuery This routine is similar to **XDMModify** except the values passed should be pointers. These pointers will be passed to the component's query function which will fill in the proper values for the named attributes. The last argument to this routine should always be XDM_END.

XDMDelete This function releases all resources consumed by the component whose dialog is *dialog* and unique identifier is *fieldId*. Like other routines, if *fieldId* is XDM_WINDOW, the dialog itself will be deleted. In general, deleting a component deletes all of its children as well.

XDMPost This routine posts *dialog* at location (*x, y*). Normally, this function returns and the programmer is expected to enter an event loop waiting for appropriate events. However, if *func* is provided, the dialog is posted as a *moded* dialog. In this case, **XDMPost** will internally handle the events associated with the window and call *func* whenever a component signals (see XDMFilter for signal details). The form of the function is:

```
int func(evt, diag, fldId)
XEvent *evt;
Window diag;
int fldId;
```

If the event is a signal from a component, *diag* will be non-zero. Otherwise, the event could not be handled by XDM and will be returned in *evt*. There are three options available when specifying *options:* XDM_INTERACT, XDM_MOUSE, and XDM_FREEZE. If XDM_INTERACT is specified, **XDMPost** will ignore the (*x, y*) parameters and interactively query the user for the dialog position. If XDM_MOUSE is specified, the dialog will be centered around the current mouse position. Finally, if *func* is provided and XDM_FREEZE is specified as an option, the routine will freeze the X server and attempt to save the pixmap under the dialog for fast restoring.

Warning: **XDMPost** handles all of the initial exposure events for a dialog and clears all events from the input queue before returning. It will discard any other user events occurring at this time.

XDMFilter This routine examines *event* and handles it if it is associated with an XDM dialog window. If the event was handled, the routine returns a positive status. If the event causes some component to generate a signal, the return code will be negative. In this case, the routine will set *dialog* and *fieldId* to the appropriate values for the component. If something went wrong (like an event not meant for XDM), the return code will be zero.

XDMEnd Unmaps *dialog* without deleting it. It can be reposted again using **XDMPost**.

B. Component Descriptions

All of the currently supported components are described in the sections that follow. These descriptions include an overview of the component, its interface description, and the message identifiers recognized by the component.

1. Dialog components.

Dialog components are the basic entity exported by XDM. Dialogs consist of an X window which may contain any number of other XDM components. Unlike other components, dialogs are identified by an X window identifier. In order to modify or query a dialog, XDM_WINDOW must be used as the fieldId to XDMModify and XDMQuery.

A dialog is also special in that it maintains a list of all components of the dialog. It is from this list that the fieldId's of other components are allocated. All other creation routines in the interface query the dialog in order to determine their fieldIds.

Dialogs also maintain carnal knowledge of type-in components. This knowledge is used to implement the concept of a currently active type-in component. Inactive type-in components pass KeyPressed events upward to the dialog which direct the events to the currently active type-in component. Thus, typing anywhere in the dialog always causes the text to be directed to the active type-in component. Initially, the first type-in component created under a dialog is considered the active component.

XDMDialogCreate

This routine creates and returns a new dialog which is a child of *parent* (usually the root window). Dialogs are recursive; a dialog can act as the parent of any number of other dialogs. If there was a problem creating the new dialog, the routine will return NULL. Dialog parameters are controlled using the standard message passing routines **XDMModify, XDMQuery,** and **XDMDelete.** The messages accepted by a dialog are shown in the tables below.

Dialog XDMModify Messages

Name	*Type*	*Description*
XDM_NAME	char *	String used in dialog icon
XDM_BGNAME	char *	Name of the background color
XDM_BGPIXEL	int	Background pixel value
XDM_HORTPAD	int	Horizontal padding (pixels)
XDM_VERTPAD	int	Vertical padding (pixels)
XDM_SMALL	int	Become as small as possible

XDM_BGNAME and XDM_BGPIXEL are mutually exclusive: one can either specify the color in standard X text format or as a pixel value previously allocated by XGetHardwareColor or XGetColorCells. Note that all string parameters passed to XDM will be copied into local storage. The padding parameters specify the minimum amount of space around the outside of the dialog; i.e. the placement of components in the dialog will be offset by this amount. If XDM_SMALL is given a non-zero value, the dialog will attempt to shrink to its smallest possible size.

Dialog XDMQuery Messages

Name	*Type*	*Description*
XDM_NAME	char **	Name of dialog
XDM_BGNAME	char **	Background color name
XDM_BGPIXEL	int *	Background pixel value
XDM_WIDTH	int *	Width of dialog (in pixels)
XDM_HEIGHT	int *	Height of dialog (in pixels)
XDM_HORTPAD	int *	Current horizontal padding
XDM_VERTPAD	int *	Current vertical padding

It is important to note that returned text values (char **) return pointers to the actual internal string used by XDM. These strings must not be modified.

2. Text components.

Text components are leaf objects which can display a single line of text in any font or color. A leaf object is a component which cannot contain any other component. Text components are also purely output fields. All user input to a text field is sent upward to its parent. These components are most often used to label other kinds of fields and to display messages of one kind or another.

XDMTextCreate

This routine creates a new text component, returning its fieldId. The parent of the component is specified by *dialog* and *id*. Normally, *id* is the fieldId of some component of *dialog*. However, if *id* is XDM_WINDOW, the text component will be created as a child of *dialog* itself. If there were problems creating the text component, the routine will return XDM_NO_ID. The modify and query messages recognized by text components are described in the tables that follow.

Text XDMModify Messages

Name	*Type*	*Description*
XDM_X	int	Upper left corner X coordinate
XDM_Y	int	Upper left corner Y coordinate
XDM_COLS	int	Number of characters in text
XDM_TEXT	char *	Text of component
XDM_FONTNAME	char *	Name of font used to display text
XDM_FONTINFO	FontInfo *	Previously opened font
XDM_BGNAME	char *	Name of background color
XDM_BGPIXEL	int	Background pixel value
XDM_FGNAME	char *	Name of text color
XDM_FGPIXEL	int	Foreground pixel value
XDM_DISABLE	int	If non-zero, grays out text
XDM_LOCKPOS	int	If non-zero, can't change position

The size of the text component window is never allowed to be less than the size of the text. The position of the component (XDM_X, XDM_Y) is relative to its parent component or dialog. If the number of columns is specified (XDM_COLS), the text component window will never be less than the size needed to display that number of average sized characters. The messages controlling font, background color, and foreground color may be specified either in name form (XDM_FONTNAME, XDM_BGNAME, XDM_FGNAME), or as a previously opened X entity (XDM_FONTINFO, XDM_BGPIXEL, XDM_FGPIXEL). The XDM_DISABLE message is used by other components who might wish to show text grayed when the input component containing the text is inactive. XDM_LOCKPOS is also used by other components to turn off the repositioning capability of the text component.

Text XDMQuery Messages

Name	*Type*	*Description*
XDM_X	int *	Upper left corner X coordinate
XDM_Y	int *	Upper left corner Y coordinate
XDM_WIDTH	int *	Width of text component
XDM_HEIGHT	int *	Height of text component
XDM_MINWIDTH	int *	Minimum width of text component
XDM_MINHEIGHT	int *	Minimum height of text component
XDM_COLS	int *	Columns of text (if any)
XDM_TEXT	char **	Text of component
XDM_FONTNAME	char **	Name of font used to display text
XDM_FONTINFO	FontInfo **	Previously opened font
XDM_BGNAME	char **	Name of background color
XDM_BGPIXEL	int *	Background pixel value
XDM_FGNAME	char **	Name of text color
XDM_FGPIXEL	int *	Foreground pixel value
XDM_DISABLE	int *	If non-zero, grays out text

All of the value fields for query messages should be pointers to the data type of the argument. The pointer should point to a space large enough to contain the value. The routine passes back the actual internal character strings. Thus, these strings should be copied if the programmer plans to change them.

3. Button components.

A button component is a region of a dialog or component that toggles its internal binary value in response to a mouse button click inside its boundaries. Buttons are heirarchical objects and can contain one other component (usually text). When a mouse button is depressed inside the button boundaries, the button component will attempt to swap the foreground and background colors of its child to indicate the change. Visually, a button appears as an outline around its child component. The foreground and background colors of the component itself also toggle with its value.

XDMButtonCreate

This routine creates a new text component, returning its fieldId. The parent of the component is specified by *dialog* and *id*. Normally, *id* is the fieldId of some component of *dialog*. However, if *id* is XDM_WINDOW, the button component will be created as a child of *dialog* itself. If there were problems creating the button component, the routine will return XDM_NO_ID. The modify and query messages recognized by button components are described in the tables that follow.

Button XDMModify Messages		
Name	*Type*	*Description*
XDM_VALUE	int	Value of button (0 or 1)
XDM_X	int	Upper left corner X coordinate
XDM_Y	int	Upper left corner Y coordinate
XDM_WIDTH	int	Width of button component
XDM_HEIGHT	int	Height of button component
XDM_USER	XDMPointer	User defined data
XDM_HILITE	int	Highlight flag (0 or 1)
XDM_SETSIGNAL	int	Arrange for signal to be generated
XDM_DISABLE	int	If non-zero, disables button operation
XDM_NOCHANGE	int	If non-zero, ignores formatting directives
XDM_LOCKPOS	int	If non-zero, prevents user repositioning
XDM_OPTWIDTH	int	Optional width formatting change
XDM_OPTHEIGHT	int	Optional height formatting change

The size of the button component is never allowed to be less than the size of its child component. The position of a button (XDM_X, XDM_Y) are specified relative to its parent component or dialog. The colors of the button are inherited from its child component. If XDM_SETSIGNAL is set, every time the button changes state, **XDMFilter** will return with a status indicating a signal has occurred. The *dialog* and *fieldId* parameters of **XDMFilter** will be set to those of the signaling button. The XDM_USER feature allows the programmer to attach his own data structures to the button and act accordingly when a signal occurs. If XDM_HILITE is non-zero, the button outline will be drawn in a way which makes the button stand out. This is often used to indicate which button of many the user should normally choose under most circumstances. If XDM_DISABLE is non-zero, the button and its child component will become "grayed out" and the button will refuse to toggle.

Buttons have a number of messages which are format related (see the description of Rows and Columns for details). These options include XDM_OPTWIDTH, XDM_OPTHEIGHT, XDM_NOCHANGE, and XDM_LOCKPOS. XDM_OPTWIDTH and XDM_OPTHEIGHT specify an "optional" width and height which is offered to the button by a formatting component. Normally, the button will always accept this size as long as it is larger than the button's child component. However, if the XDM_NOCHANGE flag is set to a non-zero value, the button will reject optional size requests and remain the same size. The XDM_LOCKPOS flag is used by formatting components to turn off position changes to a button under that component's control.

Button XDMQuery Messages

Name	*Type*	*Description*
XDM_VALUE	int *	Value of button (0 or 1)
XDM_X	int *	Upper left corner X coordinate
XDM_Y	int *	Upper left corner Y coordinate
XDM_WIDTH	int *	Width of button component
XDM_HEIGHT	int *	Height of button component
XDM_MINWIDTH	int *	Minimum width of button
XDM_MINHEIGHT	int *	Minimum height of button
XDM_USER	XDMPointer *	User defined data
XDM_HILITE	int *	Highlight flag (0 or 1)
XDM_SETSIGNAL	int *	Signal flag (0 or 1)
XDM_DISABLE	int *	If non-zero, disables button operation
XDM_NOCHANGE	int *	If non-zero, ignores formatting directives
XDM_LOCKPOS	int *	If non-zero, prevents user repositioning

For all passed pointers, the pointer should point at an area large enough for the queried value. In the case of text strings, the pointer returned is a pointer to the internal character string of the package and should not be modified.

4. Blender components.

The blender component is a special leaf component which consists of a small window containing a *check box.* This component is related to other blender components. These related blenders form a *blender set.* When a button click occurs inside a blender, the internal binary value of the blender is set and the internal state of all other blenders in the blender set are turned off. This mechanism can be used to offer the user the choice of exactly one option among many. Visually, a blender appears as a small box with rounded corners. When the internal state of a blender is set, a small check mark is drawn inside this box.

XDMBlenderCreate

This routine creates a new blender component and returns its fieldId. The parent of the component is specified by *dialog* and *id.* As with other components, if *id* is XDM_WINDOW, the component will be created as a child of *dialog.* The parameter *relId* should be the fieldId of some other previously defined blender object. The blender will be added to the blender set of the specified blender component. If the blender is the first in a blender set, *relId* should be set to XDM_NO_ID. If the new component could not be created, the routine will return XDM_NO_ID. The modify and query messages recognized by blender components are described in the tables that follow.

Blender XDMModify Messages

Name	*Type*	*Description*
XDM_VALUE	int	If non-zero, turns on this blender
XDM_X	int	Upper left corner X coordinate
XDM_Y	int	Upper left corner Y coordinate
XDM_SETSIGNAL	int	If non-zero, arrange for signal
XDM_USER	XDMPointer	User defined data
XDM_HILITE	int	Highlight flag (0 or 1)
XDM_DISABLE	int	If non-zero, blender is disabled
XDM_BGNAME	char *	Name of background color
XDM_BGPIXEL	int	Background pixel value
XDM_FGNAME	char *	Name of foreground color
XDM_FGPIXEL	int	Foreground pixel value
XDM_LOCKPOS	int	If non-zero, position is locked.

The size of a blender component is fixed and does not change. The position of the blender is specified relative to its parent component or dialog. If XDM_SETSIGNAL is set, every time the internal blender value is set, **XDMFilter** will return with a status indicating a signal has occurred. The *dialog* and *fieldId* parameters of **XDMFilter** will be set to those of the signalling blender. The XDM_USER feature allows the programmer to attach his own data structures to the blender and act accordingly when a signal occurs. If XDM_HILITE is non-zero, the blender box will be drawn in a way which makes it stand out from other blenders. This can be used to indicate which blender the user should choose under normal circumstances. If XDM_DISABLE is non-zero, the blender will become "grayed out" and the blender will refuse to activate when a mouse button is clicked inside its borders. The messages controlling blender color can be specified either by name (XDM_BGNAME or XDM_FGNAME), or by previously allocated pixel values (XDM_BGPIXEL or XDM_FGPIXEL). The XDM_LOCKPOS messages is normally used by formatting components to turn off the positioning capability of a blender under its control.

Blender XDMQuery Messages

Name	*Type*	*Description*
XDM_VALUE	int *	If non-zero, turns on this blender
XDM_X	int *	Upper left corner X coordinate
XDM_Y	int *	Upper left corner Y coordinate
XDM_WIDTH	int *	Width of blender
XDM_HEIGHT	int *	Height of blender
XDM_MINWIDTH	int *	Minimum width of blender
XDM_MINHEIGHT	int *	Minimum height of blender
XDM_SETSIGNAL	int *	If non-zero, arrange for signal
XDM_USER	XDMPointer *	User defined data
XDM_HILITE	int *	Highlight flag (0 or 1)
XDM_DISABLE	int *	If non-zero, blender is disabled
XDM_BGNAME	char **	Name of background color
XDM_BGPIXEL	int *	Background pixel value
XDM_FGNAME	char **	Name of foreground color
XDM_FGPIXEL	int *	Foreground pixel value
XDM_LOCKPOS	int *	If non-zero, position is locked.

All of the pointer items passed to the query function should point at areas large enough to receive the filled in item. The width and height of all blenders is fixed. Messages reporting this information are provided for completeness. Returned character pointers point to the internal string used by the package and should not be modified.

5. Edit region component.

An edit region component is a rectangular area for editing text. It is a composite component made up of an array of edit line components. Edit region components automatically spawn these edit line objects. Edit line components are not part of the official interface to XDM and are not described here.

The edit region component is one of the most complex as far as operation is concerned. At any one time, each top-level dialog component maintains one *active edit-region component.* This component is indicated by a cursor (a pointer under a line of text) inside the active edit-region. All keyboard input anywhere in the dialog will be directed to this component. All normal printing characters insert themselves into the edit region at the current cursor location. Edit regions do not scroll. The user is not allowed to enter more text than there is space in the edit region component. However, the programmer can make the edit region bigger in this case. Mouse clicks inside the edit region will position the cursor to that spot and insertion will continue from there. In addition, many emacs like character control sequences can be used for basic text editing operations:

Editing Features	
Key	*Description*
^A	Move to beginning of the line
^E	Move to the end of the line
^P	Move to the previous line
^N	Move to the next line
^F	Move forward one character
^B	Move backward one character
^H or <del>	Delete the previous character
^D	Delete the next character
^U or ^X	Delete the current line

There are several ways to change the currently active edit region component. First, the user can move the mouse over another edit region field and press a mouse button, thus activating that field. Second, the user can type **<tab>** and **^Q** to move to the next and previous fields respectively. The next and previous fields are determined by the order of creation of edit region components. Typing **<tab>** in the last edit region to be created causes the first edit region created to become active. Similarly, typing **^Q** in the first edit region to be created causes the last edit region created to become active. Finally, typing **^N** in the last line of an edit region is equivalent to typing **<tab>**, and typing **^P** in the first line of an edit region is equivalent to typing **^Q.**

XDMEdRegCreate

This routine creates a new edit region component and returns its fieldId. The parent of the component is specified by *dialog* and *id.* Like other components, if *id* is XDM_WINDOW, the component will be created as a child of *dialog.* If there were errors while attempting to create the component, the routine will return XDM_NO_ID. The modify and query messages recognized by edit region components are described in the tables that follow.

Edit Region XDMModify Messages		
Name	*Type*	*Description*
XDM_VALUE	char *	Value of edit region
XDM_X	int	Upper left corner X coordinate
XDM_Y	int	Upper left corner Y coordinate
XDM_CURROW	int	Cursor row
XDM_CURCOL	int	Cursor column
XDM_ROWS	int	Number of lines in region
XDM_COLS	int	Average number of characters per line
XDM_FONTNAME	char *	Name of font used to draw text
XDM_FONTINFO	FontInfo *	Previously opened font
XDM_BGNAME	char *	Name of background color
XDM_BGPIXEL	int	Background pixel value
XDM_FGNAME	char *	Name of foreground color
XDM_FGPIXEL	int	Foreground pixel value
XDM_USER	XDMPointer	User defined data
XDM_ACTIVE	int	If non-zero, edit region is active
XDM_DISABLE	int	If non-zero, edit region is deactivated
XDM_SETSIGNAL	int	If non-zero, arrange for signal
XDM_LOCKPOS	int	If non-zero, lock position

The size of the edit region is controlled by the number of rows and columns (XDM_ROWS and XDM_COLS). Because both fixed and proportionally spaced fonts are supported, the number of columns is computed based on the average size of the characters in the selected font. If XDM_SETSIGNAL is set, every time the user leaves the edit region **XDMFilter** will return with a status indicating that a signal has occurred. The *dialog* and *fieldId* parameters of **XDMFilter** will be set to those of the signalling edit region. The XDM_USER features allows the programmer to attach his own data structures to the edit region and act accordingly when a signal is detected. If XDM_ACTIVE is set to a non-zero value, the edit region will become active and a cursor will be drawn in the editing space. The active status of other edit region components are *not* affected. If XDM_DISABLE is non-zero, the edit region will become "grayed out" and the user will not be able to type text into the component. Like other components, the color and font parameters of an edit region can be specified either in text form or in X format.

Edit Region XDMQuery Messages

Name	*Type*	*Description*
XDM_VALUE	char **	Value of edit region
XDM_X	int *	Upper left corner X coordinate
XDM_Y	int *	Upper left corner Y coordinate
XDM_CURROW *	int	Cursor row
XDM_CURCOL *	int	Cursor column
XDM_ROWS	int *	Number of lines in region
XDM_COLS	int *	Average number of characters per line
XDM_WIDTH	int *	Width of edit region (in pixels)
XDM_HEIGHT	int *	Height of edit region (in pixels)
XDM_MINWIDTH	int *	Minimum width of edit region
XDM_MINHEIGHT	int *	Minimum height of edit region
XDM_FONTNAME	char **	Name of font used to draw text
XDM_FONTINFO	FontInfo **	Previously opened font
XDM_BGNAME	char **	Name of background color
XDM_BGPIXEL	int *	Background pixel value
XDM_FGNAME	char **	Name of foreground color
XDM_FGPIXEL	int *	Foreground pixel value
XDM_USER	XDMPointer *	User defined data
XDM_ACTIVE	int *	If non-zero, edit region is active
XDM_DISABLE	int *	If non-zero, edit region is deactivated
XDM_SETSIGNAL	int *	If non-zero, arrange for signal
XDM_LOCKPOS	int *	If non-zero, lock position

All of the pointer items passed to the query function should point to an area large enough to receive the entire item. The value of an edit region is returned as a null-terminated string where lines are marked by <eol>. Note that the edit region field will often break lines itself if extremely large lines are passed to it. Thus, the component does not guarantee that what comes in is what comes out (even if no changes to the text occur). All returned character pointers (including the edit region value) point to memory owned by the package and should not be modified.

6. Row/column components.

Row/column components allow the programmer to group other components into rows and columns. The programmer can then control the inter-component spacing, outside padding, pitch, and justification of these components as a unit. Row/column components can contain any number of other components of any type (including other row/column components). Thus, almost any type of formatting is possible.

A row/column component can be either a row or a column based on the value of a flag set using **XDMModify.** If the component is a row, then components created under the row will be positioned horizontally (from left to right) relative to the origin of the row. Similarly, if the component is a column, then components created under the column will be positioned vertically (from top to bottom) relative to the origin of the column. Normally, these components are placed next to each other in the order of creation and centered with respect to all other components in the row or column.

A row/column component that contains other row/column components can also be designated as an **array.** This designation causes a row/column to align each item in a sub-component with the corresponding items in other sub-components. For example, this might be used to format a list of text and blender components. First, the text and blender components could each be placed in normal columns. Then, the two columns could be placed in a row with the array designator set. This would cause each text label to line up with each blender.

Visually, row/column components have only a background color. This color will only be seen if there are portions of the row/column which are not covered by its sub-components. It is important to note that all sub-components of a row/column inherit this background color by default. Thus, it is not necessary to explicitly set the background color of sub-components of a row/column if the desired background is the same as the row/column background.

XDMRowColCreate

This routine creates a new row/column component (which is a column by default), and returns its fieldId. The parent of the component is specified by *dialog* and *id.* Similar to other components, if *id* is XDM_WINDOW, the row/column component will be created as a child of *dialog.* The messages recognized by row/column components are described in the tables that follow.

Row/Column XDMModify Messages		
Name	*Type*	*Description*
XDM_ISROW	int	If non-zero, component is a row (otherwise a column)
XDM_ARRAY	int	If non-zero, child will be arrayed
XDM_X	int	Upper left corner X coordinate
XDM_Y	int	Upper left corner Y coordinate
XDM_HORTPAD	int	Outside horizontal padding (in pixels)
XDM_VERTPAD	int	Outside vertical padding (in pixels)
XDM_HORTJUST	int	Horizontal justifcation (see below)
XDM_VERTJUST	int	Vertical justification (see below)
XDM_SPACE	int	Spacing between objects (in pixels)
XDM_BGNAME	char *	Name of background color
XDM_BGPIXEL	int	Pixel value of background
XDM_RECOMP	int	If on, causes recomputation
XDM_NOCHANGE	int	If non-zero, disallow growing
XDM_OPTWIDTH	int	Optional formatting width
XDM_OPTHEIGHT	int	Optional formatting height

The size of a row/column component is a direct function of its child components. Initially, a new row/column component is a column. If XDM_ISROW is given a non-zero value, it will change into a row. Note that if there are sub-components of a row/column when XDM_ISROW is changed, they will immediately change orientation. If XDM_ISARRAY is given a non-zero value, each item of row/column sub-components will be aligned with the corresponding items of all other row/column sub-components. The location of the row/column (XDM_X, XDM_Y) is specified relative to the parent of the component. XDM_HORTPAD and XDM_VERTPAD can be used to control the spacing from the largest items in the row/column and the outer border of the row/column. XDM_HORTJUST and XDM_VERTJUST set the alignment justification of the sub-components of a row/column. XDM_HORTJUST only has effect for columns and can only take the predefined values XDM_LEFT, XDM_CENTER, or XDM_RIGHT. XDM_VERTJUST only has effect for rows and can only take the predefined values XDM_TOP, XDM_CENTER, and XDM_BOTTOM. XDM_SPACE controls the spacing between each sub-component of a row/column. XDM_BGNAME and XDM_BGPIXEL can be used to specify the background of the row/column. As mentioned in the introduction, sub-components inherit this background unless otherwise specified.

As sub-components change in size, row/column components must recompute the placement of these components. Normally, this recomputation is done once for every change detected. This recomputation can be expensive if a large number of sub-components change in succession. To overcome this problem, XDM_RECOMP can be temporarily turned off by the programmer while extensive modifications of sub-components take place and turned back on afterward. This increases the response time of the package significantly.

The remaining options to row/column components are generally used by higher level formatting components. When a dialog expands, the top-level dialog offers the excess space to all of the top level components. This offering is done through the XDM_OPTWIDTH and XDM_OPTHEIGHT messages. Normally, row/column objects accept these offerings and pass them downward to child components who may choose to expand as well. However, if XDM_NOCHANGE is set to a non-zero value, the row/column will reject any offerings of excess space and it will remain the same size.

Row/Column XDMQuery Messages

Name	*Type*	*Description*
XDM_ISROW	int *	If non-zero, component is a row (otherwise a column)
XDM_ARRAY	int *	If non-zero, child will be arrayed
XDM_X	int *	Upper left corner X coordinate
XDM_Y	int *	Upper left corner Y coordinate
XDM_WIDTH	int *	Width of row/column
XDM_HEIGHT	int *	Height of row/column
XDM_MINWIDTH	int *	Smallest possible width
XDM_MINHEIGHT	int *	Smallest possible height
XDM_KIDS	int *	Number of children
XDM_HORTPAD	int *	Outside horizontal padding (in pixels)
XDM_VERTPAD	int *	Outside vertical padding (in pixels)
XDM_HORTJUST	int *	Horizontal justifcation
XDM_VERTJUST	int *	Vertical justification
XDM_SPACE	int *	Spacing between objects (in pixels)
XDM_BGNAME	char **	Name of background color
XDM_BGPIXEL	int *	Pixel value of background
XDM_RECOMP	int *	If on, causes recomputation
XDM_NOCHANGE	int *	If non-zero, disallow growing
XDM_OPTWIDTH	int *	Optional formatting width
XDM_OPTHEIGHT	int *	Optional formatting height

XDM_WIDTH and XDM_HEIGHT return the actual size of the row/column object including all of its sub-components. XDM_MINWIDTH and XDM_MINHEIGHT return the smallest possible size for the row/column without violating the spacing and padding rules. Note that all returned string values are pointers to the actual internal string and should not be modified.

7. Foreign window components.

Foreign window components are leaf objects which allow the programmer to integrate their own X application windows inside XDM dialog boxes. These foreign windows are treated like all other leaf components. For example, they can be formatted using row/column components. However, unlike other components, the control of this window is the responsibility of the programmer. Input events to the window are not automatically handled by **XDMFilter.** Only position related parameters are handled by **XDMModify** and **XDMQuery.** Other parameters of the window must be handled directly by the programmer.

XDMForeignCreate

This routine creates a new foreign window component. It returns the window's fieldId and its X window Id (via the *win* parameter). Like other components, the parent component is specified by *dialog* and *id.* If *id* is XDM_WINDOW, the foreign window is created as a direct child of *dialog.* Initially, the window is placed at (0,0) with respect to its parent component's coordinate system. Its initial width and length are set to *w* and *h,* and the window will have a border size of *bdrSize.* Its border and background patterns will be set to *bdr* and *bgnd* respectively. The remaining parameters to the routine are functions XDM calls when messages are passed to the foreign window versions of **XDMModify** and **XDMQuery.** XDM provides defaults for all of these functions if the programmer is not interested in intercepting these calls.

```
int minSize(win, width, height)
Window win;
int *width, *height;
```

This routine should return the minimum size for the window (including the borders). If it is not provided (zero), XDM will assume the minimum size is the initial size passed to **XDMForeignCreate.**

```
int optSize(win, width, height)
Window win;
int width, height;
```

XDM will call this function which it has found extra space for the object to expand. The width and height are guaranteed to be larger than the minimum width and height reported by **minSize.** The component may choose not to grow larger than it is currently. However, it should shrink to the given width and height if they are smaller than the current size. If this function is not provided, all optional size requests will be discarded.

```
int realSize(win, x, y, width, height)
Window win;
int *x, *y;
int *width, *height;
```

This routine should always return the current position and size of the object (including borders). This size is the one used for final formatting by the row/column component. If the function is not provided, XDM will query the server for the current size and position of the window.

```
int thePos(win, x, y)
Window win;
int x, y;
```

This routine should position the object to the specified coordinates relative to the parent component. If it is not provided, XDM will use **XMoveWindow** in this slot.

```
int delFunc(win)
Window win;
```

This routine should release any user data structures associated with the foreign window. The window itself is destroyed by XDM and need not be destroyed in the deletion function. If there are not user structures associated with the window, this function need not be provided.

Finally, whenever the foreign window changes size by some mechanism outside the control of XDM, the programmer should call the following routine:

```
int XDMForeignResize(win, oldW, oldH, newW, newH)
Window win;
int oldW, oldH;
int newW, newH;
```

The parameters *oldW* and *oldH* should contain the size of the window before the change, and *newW* and *newH* should contain the size of the window after the change. The routine sends a message to the foreign window's parent informing it of the change to it can compensate. The position of the foreign window should always be changed using **XDMModify.** All of the messages supported by the interface are described in the tables that follow.

Foreign Window XDMModify Messages

Name	*Type*	*Description*
XDM_X	int	Upper left corner X coordinate
XDM_Y	int	Upper left corner Y coordinate
XDM_OPTWIDTH	int	Optional formatting width
XDM_OPTHEIGHT	int	Optional formatting height

The position of the window (XDM_X, XDM_Y) should be specified relative to the component's parent. Changing the position of a foreign window causes XDM to call the function **thePos.** If extra space becomes available, XDM will notify the foreign window using the messages XDM_OPTWIDTH and XDM_OPTHEIGHT. This will cause the foreign window interface to call the function **optSize.**

Foreign Window XDMQuery Messages

Name	*Type*	*Description*
XDM_X	int *	Upper left corner X coordinate
XDM_Y	int *	Upper left corner Y coordinate
XDM_WIDTH	int *	Current width of window
XDM_HEIGHT	int *	Current height of window
XDM_MINWIDTH	int *	Smallest possible width
XDM_MINHEIGHT	int *	Smallest possible height

XDM calls the function **realSize** in order to service requests for the messages XDM_X, XDM_Y, XDM_WIDTH, and XDM_HEIGHT. For the messages XDM_MINWIDTH and XDM_MINHEIGHT, XDM consults the function **minSize.**

8. Utility functions.

XDM provides some general purpose routines for searching through the components of a dialog, determining the dialog of a component, and querying the type of a component. These routines are described below:

XDMForEach This routine calls *func* once for each component in *dialog.* The form of *func* is:

```
int func(diag, fldId, type, arg)
Window dialog;
int fieldId;
int type;
XDMPointer arg;
```

The *diag* and *fldId* parameters identify the field. For example, these parameters can be used as arguments to **XDMModify** or **XDMQuery.** *type* is the component type (see **XDMTypeQuery** for a list of component types). The parameter *arg* is the same as that passed to **XDMForEach.** It can be used to pass state information to *func.*

XDMFindDialog All components in XDM are actually X windows. This routine returns the dialog of a component given its X window identifier. If no such dialog exists, the routine returns NULL.

XDMTypeQuery This routine returns the type of a component given *dialog* and *fieldId.* The valid types currently supported by XDM are:

Component Types	
Name	*Description*
XDM_DIALOG_OBJ	Dialog component
XDM_TEXT_OBJ	Text component
XDM_BTN_OBJ	Button component
XDM_BLEND_OBJ	Blender component
XDM_EDREG_OBJ	Edit region component
XDM_ROWCOL_OBJ	Row/column component
XDM_FOREIGN	Foreign window

9. Default handling.

All parameters of all components have default values.

DialogFont	The default font used for all components with font parameters. If not specified, XDM uses "6x10".
DialogBackground	This parameter specified the default background color for all components with background attributes. The default color is white.
DialogForeground	This default specifies the default foreground color for all components with foreground parameters. If not specified, XDM uses black.
DialogBorderWidth	This default specifies the border width of all dialogs. The default border width is 2 pixels.
DialogBorderColor	This parameter specifies the color of all dialog borders. The default color is black.
DialogCursorColor	This parameter controls the color of the mouse cursor when it is inside a XDM dialog box. By default, the color used is black.

Appendix 5. iv: Change Values of Variables Interactively *

A. General Information

1. Synopsis.

#include "X/Xlib.h"
#include "iv.h"

2. Routines.

Window iv_init(programName, title, maxValChars) char *programName; char *title; int maxValChars;

iv_addIVar(docString, varP, type, precision, func) char *docString; int *varP; int type; int precision; iv_PFI func;

iv_updateIVar(varP) int *varP;

iv_WhichSelect(itemPtr) ivWhichItem *itemPtr;

int iv_promptIVar(varP, bell) int *varP; int bell;

int iv_MapIVWindow(x, y, option) int x, y; int option;

int iv_handleEvent(theEventP) XEvent *theEventP;

void iv_SetEraseFlag(status) int status;

void iv_SetEnableFlag(varP, status) int *varP; int status;

int iv_processAllEvents();

int iv_destroy()

3. Overview.

IV is an interactive forms-based input system for the X Window System. It provides the means for displaying and controlling variables in a C program through the use of an X window, while the program is running. All the interactive variables are shown on a window, one on each row. Each variable is displayed with its description and an edit region containing its current value. At any one time, the IV window maintains at most one *active* edit region, where the variable may be changed. All keyboard input anywhere in the IV window will be directed to this region. Edit regions are activated by placing the mouse cursor over an edit region, and either clicking a mouse button or pressing a key. *Changes are accepted only by a carriage return or end-of-file.*

* Program written by Andrea Casotto and Benjamin Chen, University of California, Berkeley.

For integer or floating-point variables, two buttons are provided to change the value of the variable. The "+" button has the following effect: If the *LEFT* mouse button is pressed, the value of the variable is incremented by 1%, or by one for integer variables. If the *MIDDLE* mouse button is pressed, the value of the variable is incremented by 10% If the *RIGHT* mouse button is pressed, the value of the variable is doubled.

The "-" button has similar behavior, but the value of the variable is decremented. For IV_WHICH variables, the plus and minus buttons advances or reviews through a list of values specified by the user. (See test program and iv.h to see how to implement an IV_WHICH variable.) For boolean variables, one button is provided for easy toggling of its state.

B. Description of Routines

iv_Init This procedure initializes the IV package and reads the user's ~/.Xdefaults file. *programName* is the name of the program (normally argv[0]). The defaults for the package are listed in the defaults section. *title* is the name of the IV window. The name is centered at the top of the window. *maxValChars* specifies the maximum number of characters allowed in the edit region of the variables, except for string variables, which hold more depending on the font size. The procedure returns the window ID of the IV window it was successfully created. If it was not successful, it will return a null window ID. A connection to an X display must have been established before calling this routine. Only one window can be open at any time.

iv_addIVar This routine adds an IV variable to the window. The variable will not be displayed until iv_MapIVWindow is called. *docString* should contain a short documentation (usually the variable name) for the variable. The IV window will resize according to the longest documentation string it is given. *varP* should be the address of the variable to be edited. To make lint happy, a (int *) should precede the variable address, since IV always casts the pointer to an (int *). *type* should specify the type of variable to be added.

There are six types of IV variables: IV_DOUBLE, IV_INT, IV_BOOLEAN, IV_STRING, IV_TOGGLE, and IV_WHICH. Except for IV_TOGGLE, each type has a documentation field and an edit region. An edit region is indicated by the appropriate background color specified in the user's XDefaults. Typing "MAXINT" in the edit region will display the maximum integer allowed for integer types. Similarly, "MAXFLOAT" displays the maximum floating point number, while "Infinity" or "HUGE" will set the variable to the IEEE standard infinity (99.e999), or MAXFLOAT if no such definition exist on the machine. IV_DOUBLE, IV_INT, and IV_WHICH variables have plus and minus buttons. IV_TOGGLE and IV_BOOLEAN variables are basically identical except that IV_TOGGLE does not display the state of the variable. They will both set the variable to 1 or 0 , respectively. IV_BOOLEAN variables will and show "TRUE" or "FALSE" in their edit regions. IV_STRING variables have no associated button.

precision specifies the number of places to the right of the decimal to print a floating point number. Numbers greater than 999999.9 and less than 0.001 are shown in exponential notation. IV_NO_OPT should be used for all other variable types. *func* is a pointer to a function that returns an integer. The function should take no parameters, and will be called whenever its button is pushed. The function should return an IV_OK when successful. If no routine is to be called, IV_NULL_FUNCTION should be used. *iv_PFI* is defined in iv.h.

iv_updateIVar	This routine prints the current value of the variable pointed to by *varP*. in the IV window. It should be called when the program internally changes the value of the given variable, and the user wants to see the new value.
iv_WhichSelect	This routine returns the index of the item selected from the list pointed to by *itemPtr.* The first item is item #0.
iv_promptIVar	This routine prompts the user to edit the value of a variable. Unlike the normal editing feature it halts the process running iv, since it enters its own internal loop. No other action can be taken until the prompted variable is accepted by a carriage return or EOF. The IV window is automatically raised upon prompting. The prompted variable is specified by *varP*, and is highlighted. *bell* specifies the volume of the bell (0-7).
iv_MapIVWindow	This routine maps the current IV window at location (x, y). There are three options available when specifying *option:* IV_INTERACT, IV_MOUSE, and IV_NO_OPT. If IV_INTERACT is specified, the routine will ignore the (x, y) parameters and interactively query the user for the IV window position. If IV_MOUSE is specified, the window will be centered around the current mouse position.
iv_handleEvent	This routine will return *IV_EXTRANEOUS_EVENT* if the event in the argument is not affecting any of the interactive variables. Otherwise, it returns *IV_OK* after processing the event.
iv_SetEraseFlag	This routine sets to the erase flag described in the EraseValue default to *status.* If non-zero, EraseValue will be "on."
iv_SetEnableFlag	This routine sets to the enable flag for the specified variable to *status.* The enable flag determines whether a variable can be edited or changed. If the enable flag is zero, the background of the type in field, if one exists, will be changed to the background of the IV window. Also, any buttons will disappear.

Iv_processAllEvents	This routine processes ALL events. It removes the need for an event loop in the main program. *This procedure should be used only when IV is the only X application the program is running.*
Iv_destroy	This routine unmaps and destroys the IV window and frees all resources associated with IV.

C. Defaults

IV has a number of parameters that can be set using the .Xdefaults file. The format should be <program name>.iv.<default>.

Background	Set the background color. Default is light grey on color displays, black on monochrome.
BorderColor	Set the border color. Default is black on color displays, white on monochrome.
BorderWidth	Set the border width of the main IV window, and the border around the edit region windows. Default is 1.
ButtonColor	Set the color of the buttons. Default is yellow on color displays, white on monochrome. For best results, choose a non-dark color.
CursorColor	Set the color of the mouse cursor. Default is green on color displays, white for monochrome.
EditBackground	Set the background color of the edit region. Default is light blue on color displays, black for monochrome.
EditFont	Specify the font to print the edit region. Default is 6x10.
EditFontColor	Set the font color of the edit region. Default is red for color displays, white for monochrome.
EraseValue	If "on" clear the edit region upon editing the variable. The default is "off." Note that data can still be recovered by CONTROL_U.
Padding	Specifies the extra padding above and below each IV row (text and variable). The default is 2.
TextFont	Specify the font to print the documentation field. Default is 6x10.
TextFontColor	Set the font color of the documentation field. Default is blue for color displays, white for monochrome.

TitleFont	Specify the font to print the title. Default is 9x15.
TitleFontColor	Set the font color of the title Default is dark slate blue for color displays, white for monochrome.

D. Sample Program

```
/* test the iv.a routines */

#include <X/Xlib.h>
#include "iv.h"

extern char *strcpy();
extern char *calloc();

int
modifiedN()
{
  (void) printf("N has been modified0);
  return(IV_OK);
}

static ivWhichItem itemPtr[] = {{"Simple", 0},
                {"Complex", 0},
                {"Net", 0},
                    {"Pin", 1}};


main()
{
  double a, c, d;      /* test variables */
  int   n, b;
  XEvent theEvent;
  char *getenv();
  char *displayName;
  Display *display;
  char *theString;
  int   exitFlag = 0;
  int   selected;

  theString = (char *) calloc((unsigned) (20), sizeof(char));
  (void) strcpy(theString, "Test String");
```

```
displayName = getenv("DISPLAY");
 if ((display = XOpenDisplay(displayName)) == (Display *) 0 ) {
   abort();
 }

 a = 100; c = 12.67; d = 19999999e12;
 n = 13;
 b = 0;

 (void) iv_init("test","Test interactive variables", 12);
 iv_addIVar("WhichOne", itemPtr, IV_WHICH,
       (sizeof(itemPtr)/sizeof(ivWhichItem)) , IV_NULL_FUNCTION);
 iv_addIVar("Variable A", &a, IV_DOUBLE, 1, IV_NULL_FUNCTION);
 iv_addIVar("Variable C", &c, IV_DOUBLE, 2, IV_NULL_FUNCTION);
 iv_SetEnableFlag(&c, 0);
 iv_addIVar("Variable D", &d, IV_DOUBLE, 3, IV_NULL_FUNCTION);
 iv_SetEnableFlag(&d, 0);
 iv_addIVar("Integer N" , &n, IV_INT, IV_NO_OPT, modifiedN);
 iv_addIVar("Boolean var", &b, IV_BOOLEAN,
       IV_NO_OPT, IV_NULL_FUNCTION);
 iv_addIVar("string", &theString, IV_STRING,
       IV_NO_OPT, IV_NULL_FUNCTION);
 iv_addIVar("Exit IV", &exitFlag, IV_TOGGLE,
       IV_NO_OPT, IV_NULL_FUNCTION);
 iv_MapIVWindow(IV_NO_OPT, IV_NO_OPT, IV_MOUSE);

 while (!exitFlag) {
     XNextEvent(&theEvent);
     if (iv_handleEvent(&theEvent) != IV_OK) {
     (void) printf("Extraneous event0);
     }
 }

 iv_addIVar("Duplicate variable A", &a, IV_DOUBLE,
       IV_NO_OPT, IV_NULL_FUNCTION);
 iv_addIVar("Duplicate variable C", &c, IV_DOUBLE,
       IV_NO_OPT, IV_NULL_FUNCTION);
 iv_SetEnableFlag(&c, 1);
 iv_MapIVWindow(IV_NO_OPT, IV_NO_OPT, IV_NO_OPT);
 a = 9.6;
 d = 9.9;
 iv_updateIVar(IV_NO_OPT);
 iv_promptIVar(&n, 1);
 iv_SetEraseFlag(1);
 c = 6.9;
 iv_updateIVar(&c);
```

```
    exitFlag = 0;
    while (!exitFlag) {
        iv_processAllEvents();
    }
    selected = iv_WhichSelect(itemPtr);
    iv_destroy();
    (void) printf("Selected : %d0, selected);
    (void) printf("Variable A: %lf0, a);
    (void) printf("Variable C: %lf0, c);
    (void) printf("Variable D: %lf0, d);
    (void) printf("Integer N: %d0, n);
    (void) printf("Boolean var: %d0, b);
    (void) printf("string:'%s'0, theString);
    (void) printf("exit flag: %d0, exitFlag);
}
```

Files: *~cad/lib/libiv.a*

~cad/include/iv.h

Bugs: Mail complaints to Andrea Casotto or Benjamin Chen, Dept. of EECS, University of California, Berkeley, CA 94720.

Appendix 6. scrollText: Multi-font scrollable text windows for X *

A. General Information

1. *Synopsis.*

#include <X/Xlib.h>
#include "scrollText.h"

2. *Routines.*

int TxtGrab(textWin, program, mainFont, bg, fg, cur) Window textWin; char *program; FontInfo *mainFont; int bg, fg, cur;

int TxtRelease(w) Window w;

int TxtAddFont(textWin, fontNumber, newFont, newColor) Window textWin; int fontNumber; FontInfo *newFont; int newColor;

int TxtWinP(w) Window w;

int TxtClear(w) Window w;

int TxtWriteStr(w, str) Window w; char *str;

int TxtJamStr(w, str) Window w; char *str;

int TxtRepaint(w) Window w;

int TxtFilter(evt) XEvent *evt;

3. *Overview.*

The *scrollText* package implements a multi-font, multi-color, scrollable text window abstraction which runs over the X Window System. The package supports any number of windows each with its own scroll bar and character buffer. Each window can have up to eight fonts loaded. A color may be specified for each loaded font. The fonts can be mixed freely using a change font character control sequence. The scrolling operations supported are scroll relative to scroll bar, line to top, and top line to here. The size of the character buffer for each window is limited only by the process space of the controlling program.

* Program written by David Harrison, University of California, Berkeley.

B. Description of Routines

TxtGrab Takes control of a previously created window, *textWin*, and makes it into a scrollable output window. The string *program* is used to look up X defaults for the package (see section *X Defaults*). The parameter *mainFont* is the initial font used for drawing text in the window. This font is loaded into slot zero. *TxtGrab* assumes this record is fully filled (including the width table). The X library routine *XOpenFont* can be used to obtain fully filled font record structures. Additional fonts can be loaded using *TxtAddFont* (described below). The pixel value *fg* will be used to draw the scroll bar and is also used as the initial color for *mainFont*. The pixel value *bg* will be used as the background for drawing all text. *bg* is also used as the background for the scroll bar subwindow. The color of the text cursor is set to *cur*. In order for the text window to work properly, the programmer must select *ExposeRegion* and *ExposeCopy* events on the window in addition to any other events the programmer might wish to register. The routine returns a non-zero value if the window was sucessfully grabbed.

TxtRelease Releases control of a previously grabbed window. All resources consumed by the text window package are reclaimed. The window itself is *not* destroyed.

TxtAddFont Loads a new font so that it can be used in a previously grabbed text window. The parameter *fontNumber* is used to specify the slot for the new font. There are eight font slots numbered 0 through 7. If there is already a font in the specified slot, it will be replaced with the new one and an automatic redraw of the screen contents will take place. See *TxtWriteStr* and *TxtJamStr* for details on using multiple fonts. The pixel value *newColor* specifies the foreground color for the font. If TXT_NO_COLOR is specified, the color will default to the foreground color supplied when the window was grabbed. The programmer can change just the color of a font by specifing a null font for a given slot. The routine returns a non-zero value if the font was sucessfully loaded.

TxtWinP Returns a non-zero value if the specified window has been previously grabbed using *TxtGrab*. If it is not a text window, the routine returns zero.

TxtClear Clears the specified window of its contents and resets the current writing position to the upper left hand corner of the screen. The routine also *clears the contents of the text window buffer and resets the scroll bar.* The routine returns zero if the window is not a text window. This procedure should be used *instead* of the X library call *XClear.*

TxtWriteStr Writes a null-terminated string into the specified text window. The text is always appended to the end of the text buffer. If the scroll bar is positioned such that the end of the text is not visible, an automatic scroll to the bottom will be done before the text is appended. Non-printable ASCII characters are ignored. The newline character (\n) causes the current text position to advance one line and start at the left. Tabs are not supported. Lines too wide to fit on the screen will be wrapped to the next line and a line wrap indicator will be drawn in the right margin. Backspace deletes the previous character. It will do the right thing if asked to backspace past a normal or wrapped line marker. A new text font can be specified using the sequence control-F followed by a digit. The digit must be 0, 1, 2, or 3. The directive will be ignored if there is no font loaded in the specified slot. If there is no more space at the bottom of the screen, the window will scroll to make room. The routine will return zero if it could not append the text.

TxtJamStr Is identical to *TxtWriteStr* except the current screen position is *not* updated. This routine should be used if the programmer wants to append text to the buffer without causing the window to scroll. After the text has been added, the programmer should call *TxtRepaint* to update the screen contents.

TxtRepaint Redraws the specified scrollable text window. The routine repaints the entire window including the scroll bar. NOTE: for handling exposure events, *TxtFilter* should be used.

TxtFilter Handles events associated with scrollable text windows. It will handle *ExposeRegion* and *ExposeCopy* events on the main window, and *ExposeWindow* and *ButtonReleased* events in the scroll bar. It does *not* handle any other events. If it does not want to handle the event, the routine will return zero. A call to this routine should be included in the main event loop of the programmers controlling program.

C. User Interface

The *scrollText* package supports user controlled browsing through a buffer built using *TxtWriteStr* or *TxtJamStr.* Along the right side of the window is a *scroll bar* window. The scroll bar window displays a filled square representing the relative position through the buffer and the relative amount of the buffer currently on the screen. Scrolling is controlled by clicking mouse buttons in the scroll bar.

This package supports three scrolling operations: scroll to spot, line to top, and top to here. The **middle button** is used to select scroll to spot. This operation causes the screen to scroll such that the center of the scroll bar indicator moves to the current position of the mouse. This is used to scroll to a relative spot in the buffer. Line to top and top to here operations are for scrolling down or up some proportion of the screen. The **left button** selects line to top. This operation causes the screen to scroll such that the line adjacent to the mouse position becomes the top line of the screen. Thus, clicking near the top of the scroll bar scrolls only a couple of lines while a click near the bottom will scroll almost an entire screen. The **right button** is used for the top to here command. This function causes the top line of the screen to scroll down to the current position of the mouse. This allows the user to scroll up and down by the same amounts if the mouse position is kept constant.

Defaults: The current version of the library reads one default: **JumpScroll** If on, the line to top and top to here operations will not scroll to the target position smoothly. Instead, the window will be repainted once at the correct spot.

Files: *libScroll.a* (Scrollable text library)

See also: Xlib - C Language X Interface, X(8C)

Bugs: Sometimes when the window is resized, the scroll bar is repainted without a border. The origin of this bug is unknown but a work-around is to iconify and deiconify the window forcing a complete redraw.

Loading large files with many font changes is slow. Unfortunately, there is simply a lot of work which must be done. Resizing windows with extrodinarily large buffers may also take some time (the line breaks must be recomputed).

References

1. N.P. Chen, C.P. Hsu, and E.S. Kuh, "The Berkeley Building-block (BBL) Layout System for VLSI Design," in *Dig. Tech. Papers, IEEE Int. Conf. on Computer-Aided Design*, pp. 40-41, 1983.

2. W. Dai, M. Sato, and E.S. Kuh, "A Dynamic and Efficient Representation of Building Block Layout," *Proc. 24th Design Automation Conf.*, pp. 376-384, 1987.

3. W. Dai and E.S. Kuh, "Global Spacing of Building Block Layout," *Proc. VLSI Conf.*, pp. 161-173, 1987.

4. W. Dai, E.S. Kuh, "Simultaneous Floorplanning and Global Routing for Hierarchical Building-Block Layout," *Proc. Int. Conf. on Computer-Aided Design*, pp. 828-837, 1986.

5. B. Eschermann, W. Dai, E.S. Kuh, and M. Pedram, "Hierarchical Placement for Macrocells," *Proc. Int. Conf. on Computer-Aided Design*, pp. 460-463, 1988.

6. M. Khellaf "On the Partitioning of Graphs and Hypergraphs," Ph.D. Diss., *Dept. IEOR, Univ. of California, Berkeley*, 1987.

7. J.K. Ousterhout, "Corner Stitching: A Data Structure Technique for VLSI Layout Tools," *IEEE Trans. on Computer-Aided Design*, vol. CAD-3, no. 1, 1984.

8. M. Marek-Sadowska, "Route Planner for Custom Chip Design," *Dig. Tech. Papers, IEEE Int. Conf. on Computer-aided Design*, pp.246-249, 1986.

9. W.M. Dai, T. Asano and E.S. Kuh, "Routing Region Definition and Ordering Scheme for Building Block Layout," *IEEE Trans. on Computer-Aided Design*, vol. CAD-4, no. 3, pp.189-197, 1985.

10. H.H. Chen and E.S. Kuh, "Glitter: A Gridless Variable-Width Channel Router," *IEEE Trans. on Computer-Aided Design*, vol. CAD-5, no. 4, pp. 459-465, 1986.

11. H.H. Chen, "Routing L-Shaped Channels in Nonslicing Structure Placement," *Proc. of 24th Design Automation Conf.*, pp. 152-158, 1987.

12. X.M. Xiong and E.S. Kuh, "Nutcracker: An Efficient and Intelligent Channel Spacer," *Proc. of 24th Design Automation Conf.*, pp. 298-304, 1987.

13. X.M. Xiong and E.S. Kuh, "The Constraint Via Minimization Problem for PCB and VLSI Design," *Proc. 25th Design Automation Conf.*, pp. 573-578, 1988.

14. R. Dutta and M. Marek-Sadowska, "Automatic Sizing of Power/Ground (P/G) Networks in VLSI," to appear in *Proc. 26th Design Automation Conf.*, 1989.