SIMPL-DIX
(SIMulated Profiles from the Layout—Design Interface in X)

by

Hsi-Cheng Wu

Memorandum No. UCB/ERL M88/13

January 1988

SIMPL-DIX
(SIMulated Profiles from the Layout—Design Interface in X)

by

Hsi-Cheng Wu

Memorandum No. UCB/ERL M88/13

January 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SIMPL-DIX
(SIMulated Profiles from the Layout—Design Interface in X)

by

Hsi-Cheng Wu

Memorandum No. UCB/ERL M88/13

January 1988

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

SIMPL-DIX

(SIMulated Profiles from the Layout - Design Interface in X)

Hsi-Cheng Wu

Electronics Research Laboratory
Department of Electrical Engineering and Computer Science
University of California, Berkeley, California 94720

ABSTRACT

SIMPL-DIX (SIMulated Profiles from the Layout - Design Interface in X) is an X-window based interactive computer-aided-design tool for running process and device simulators.

The current version of SIMPL-DIX invokes SIMPL-2 to create the cross-sectional profile of a device along an arbitrary cut-line drawn on the layout. It also provides a number of internal tools such as HUNCH to assist the designer in examining potential problems associated with a device topography.

In this report, the internal structure of SIMPL-DIX is explored in detail. Issues regarding data structures, display interface, and application interface are examined. Information is provided to assist future extensions for incorporating additional simulation tools.

December 15, 1987

Acknowledgements

I would like to thank Professor Andrew R. Neureuther for his valuable guidance and encouragement. His time and patience throughout this project are greatly appreciated. I would also like to thank Professor William G. Oldham for his valuable comments and reading the paper.

I also gratefully acknowledge the support from my colleagues; I thank William Bell, Simon Koh, Ed Scheckler and Alex Wong for their valuable help.

To the friends who have encouraged and guided me throughout my stay at Berkeley, I thank Jarrett Lu, Kit-Ming Mak, Allen Wei, and Paul Yang for their support and friendship.

Finally, to my parents, I am deeply thankful for their constant loving support and infinite patience during my graduate years.

The financial support from SRC is gratefully acknowledged.

Table of Contents

Chapter 1
Introduction

1.1. Background

SIMPL (SIMulated Profiles from the Layout) is a family of computer-aided-design tools which simulate the topography of an integrated circuit using the layout information. Along an arbitrary cut-line drawn on the layout, SIMPL generates the evolution of the cross-sectional profile according to the process steps given to SIMPL interactively or in batch mode.

The first program of SIMPL, *SIMPL-1* [1], employs a linked-rectangular database to approximate the cross-sectional structure. It saves the simulated data in a CIF file, which can be viewed later by layout graphics editors such as KIC. With its fast simulation capability, SIMPL-1 provides a rapid on-line visual feedback on the device features to circuit designers.

The next generation of SIMPL, *SIMPL-2* [2], applies extensive two-dimensional process models to simulate the cross-sectional profile. It couples with advanced process simulators such as *SAMPLE* [3] to provide a refined topographical structure for process evaluation. Based on its linked-polygonal and grid type databases, SIMPL-2 is capable of displaying the two-dimensional process effects such as the *bird's beak*, *lateral diffusion*, *undercut in etching*, and *sidewall coverage in deposition*.

1.2. Overview

As more process and device simulation programs appear, an effort to integrate the many dissimilar tools is needed. Preserving the objective of SIMPL, *SIMPL-DIX* is developed as a high-level design interface to generate the device topography using the extracted layout information. As an X-window based CAD tool for controlling external process and device simulators, SIMPL-DIX provides the initial structure for a future integrated CAD system on a workstation environment.

The current version of SIMPL-DIX invokes SIMPL-2 to create the database needed for the cross-sectional profile. The data generated by SIMPL-2 can later be used as the *seed* for extensive simulations with external simulators. Future SIMPL-DIX release will incorporate rigorous simulation tools such as *CREEP* [4] with interface through a profile interchange format (PIF).

Unlike other SIMPL programs, SIMPL-DIX has a number of internal tools to assist the designer in running simulations. An application display editor is provided for the user to define a selected layer and to magnify a selected region. A pattern editor is included which allows the user to add or delete patterns, to modify formats of patterns, and to update pattern specifications. Finally, *HUNCH* is implemented to allow the designer to use operations between masks or sets of masks to highlight locations where topographical problems are anticipated; these critical areas can then be examined using simulators such as SIMPL-2 for process verification.

1.3. General Structure

As a high-level design tool, SIMPL-DIX contains many interfaces. A *display* interface controls the interaction of the program with the X window system; it manages the visual appearance of the application as commanded by the user. An *application* interface regulates the communication between SIMPL-DIX and external simulators; it performs transmissions and transformations of data among tools for a specified application. Finally, the *design* interface is the sum total of all display and application interfaces; it provides the necessary integration approach for a composite CAD system.

This report discusses the methodology for the design of these interfaces. Detailed descriptions of the internal data structures are provided to assist future extensions. Chapter 2 describes the low-level SIMPL-DIX display structures based on the X-window system. Chapter 3 discusses the application interface with SIMPL-2; detailed implementation techniques are pro-

vided to assist additional interfaces with external tools. Chapter 4 describes the internal mask-operation structures used by *HUNCH*. Finally, chapter 5 presents an overview of the future directions for SIMPL-DIX. The appendices at the end include the manual page and the routine catalog of SIMPL-DIX.

Chapter 2
Display Interface with X

2.1. Introduction

X is a window system that runs under 4.3BSD UNIX, ULTRIX-32, VAX/VMS, and several other operating systems [5]. It provides high-performance graphics to computers with bitmap displays, and through a library of low-level C routines, it allows application programs interacting with the window system to be built.

The following sections describe the various SIMPL-DIX display data structures based on protocol version 10 of the X window system. Defaults and command options used to control the SIMPL-DIX window are summarized in Appendix A.

2.2. Window and Viewport Management

In a window system, a *window* is an area on the display screen associated with an application; a *viewport* is a pane of a window into which an application maps output [6].

Two kinds of windows are supported by X, opaque and transparent [7]. An *opaque* window has borders and a background pattern; it obscures windows underneath it for the purpose of both input and output. A *transparent* window, on the other hand, is always invisible on the screen; it obscures other windows for the purpose of input only. These window structures are defined in the system library header file *<X/Xlib.h>*.

To exploit distinctions between these two window types, the main window associated with each invocation of SIMPL-DIX is designated by an opaque window, whereas viewports within the main window are represented by transparent windows. Since most graphics routines manipulate data at the viewport level, this approach provides structural screen references without devitalizing the main window. Prescribing viewports as transparent windows also enables different display resources such as mouse cursors to be associated with different viewports.

Each of the windows employed by SIMPL-DIX is identified with a window descriptor *Window* assigned by the X server.

The SIMPL-DIX window is divided into six major viewports: the message viewport, the command menu viewport, the left and right pattern viewports, and the upper and lower application viewports. In addition, a title-bar viewport is provided for the display of SIMPL-DIX application name; associated with each pattern viewport, a scroll-bar viewport is used to control scrolling of the display contents. The relative positions of these viewports within the main window are illustrated in Figure 2.1.

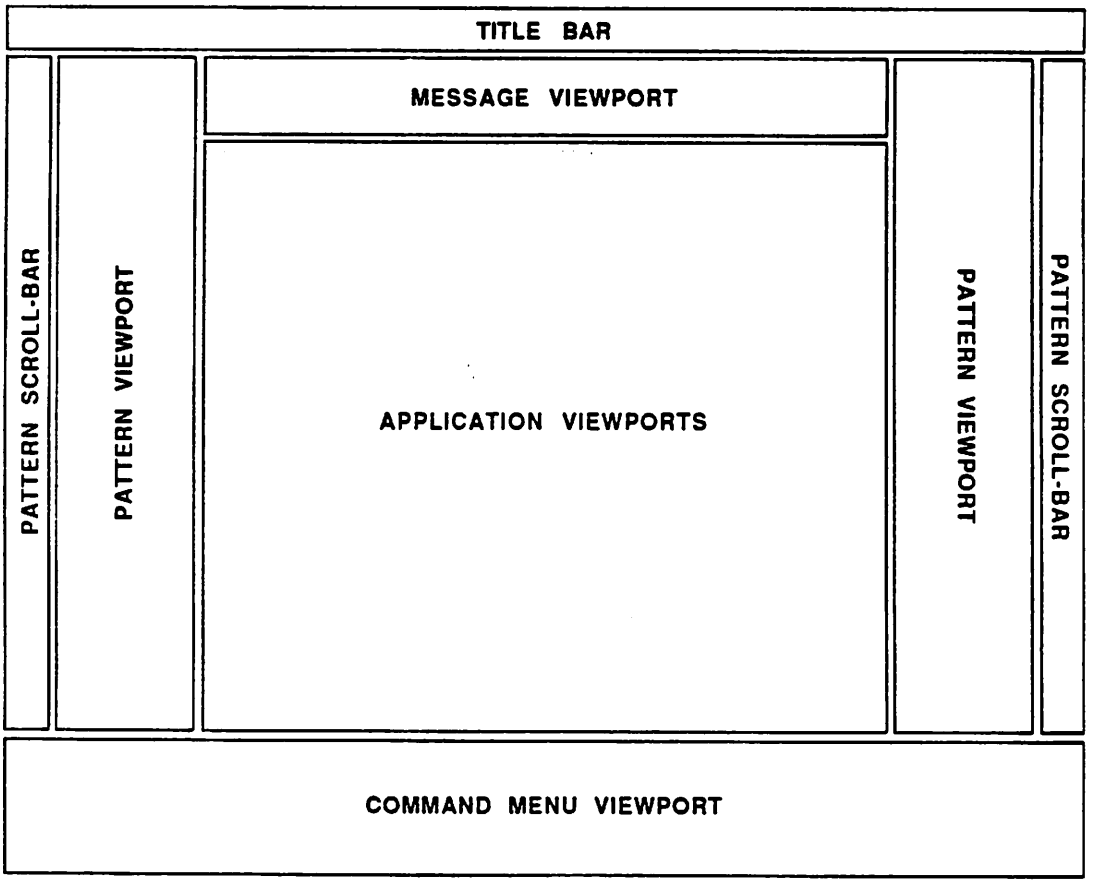


Figure 2.1: SIMPL-DIX Viewports

SIMPL-DIX maintains an internal viewport manager to control the display contents of these viewports. At each level in the program, a variety of information has to be displayed for various effects, and upon termination of a task, individual viewport has to be restored to its original state. This display control is accomplished through manipulations of a number of global flags along with procedures such as *UpdateLayoutDisplay()* and *UpdateProfileDisplay()* in the *display_control.c* source file.

The viewport manager is also responsible for handling input events passed to SIMPL-DIX by the X server. The basic event types are defined in the system library header file *<X/X.h>*. Internally, each routine waiting for an event to occur relies on the SIMPL-DIX procedure *GetEvent()* in the file *display_control.c* to return an event type. Basic tasks such as resizing the viewports and redrawing the display are performed by the viewport manager routines before an event can actually be passed to the requested procedure.

Each window event passed to a display routine is mapped to an internal SIMPL-DIX event structure, defined in the *display.h* file as follows:

```
/*
 * SIMPL-DIX event structure.
 */
typedef struct {
        Window window_id;          /* Viewport window id. */
        char key;                  /* Pressed key. */
        int button;                /* Pressed button. */
        int x, y;                  /* Pressed coordinates. */
} dixEvent;
```

The *window_id* identifies the viewport within which the event occurred. If a keyboard event has been generated, the field *key* stores the pressed character after the keyboard mapping operation. If a button event has been initiated, the field *button* identifies the mouse button that is pressed or released.

Fields *x*, *y* denote the location of the mouse cursor in the main window coordinates with which the event has occurred. Within a window, the coordinate system adopted is measured in pixels relative to the upper-left corner of the window. Each viewport has its own coordinate

system; to transform location from the main window coordinates to that in a specified viewport, macros *Viewport_X()* and *Viewport_Y()* in the *display.h* file are provided.

2.3. Command Menu Structure

SIMPL-DIX is designed with a hierarchical menu system which allows users to view the choices available to them at any one time without having to remember command words or special keys. Nine different menus are adopted; detailed descriptions for commands within each menu are listed in *SIMPL-DIX User's Guide*.

The basic command menu structures are defined in the *command.h* file as follows:

```
/*
 * SIMPL-DIX command menu descriptors.
 */
typedef struct {
        char *name[3];                  /* Entries for the command. */
        int (*proc)();                  /* Pointer to action procedure. */
} dixCommand;

typedef struct {
        int last;                       /* Previous menu id. */
        int size;                       /* Command size. */
        dixCommand *command;            /* Pointer to command list. */
} dixMenu;
```

Each command in a menu is represented by a set of text strings *name*, restricted to be up to three levels. Associated with each command, a pointer to an action procedure *proc* maintains the function for the command to perform. For convenience, all SIMPL-DIX action procedures are collected in the *dix_action1.c* and *dix_action2.c* source files.

Within the menu structure, the field *last* identifies the previous menu before the current command menu is invoked; the field *size* stores the number of commands that are allocated for the current menu. Finally, the *command* denotes the pointer to the root of the command list for the current menu.

To allow future enhancement, a number of initialization procedures are provided in the source file *command_control.c* as follows:

```
SetMenu(menu_id, menu_size)
        int menu_id, menu_size;

SetCommand(menu_id, command_id, name_0, name_1, name_2, proc)
        int menu_id, command_id;
        char *name_0, *name_1, *name_2;
        int (*proc)();
```

SetMenu() creates and initializes a menu structure. The specified *menu_size* denotes the number of commands to be allocated. The menu created is identified by the specified *menu_id*.

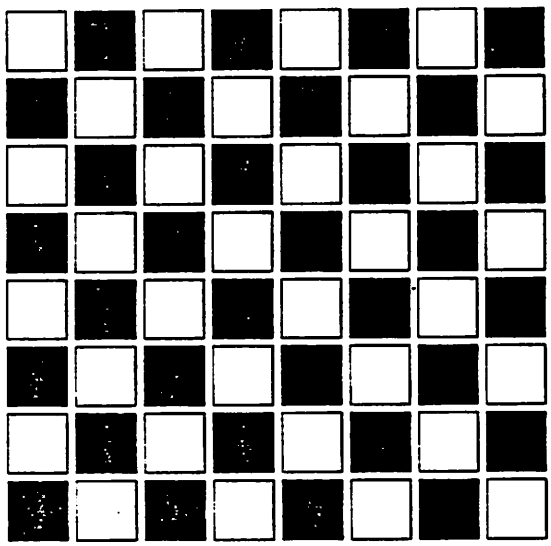
SetCommand() initializes the command entries as specified by the *names*. The command structure defined is placed on the command list for the menu specified by *menu_id*. The specified action routine *proc* is linked with the command.

To add new commands, first change constant definitions such as menu id and menu size specified in the *command.h* header file, modify the procedure *InitMenu()* in the file *command_control.c* to reflect these changes, and then include proper action procedures and structures in the program.

2.4. Color and Pattern Representations

SIMPL-DIX utilizes a user defined pattern file to create the stipple patterns needed. To maintain consistency with SIMPL-2, the *MFB* specification with 8x8 bit format is used. To allow more patterns to be defined, a 16x16 bit format is also adopted. Internally, SIMPL-DIX will check the type of pattern specifications actually use. Formats for specifying color patterns are illustrated in Figure 2.2.

To allow the user to modify the existing pattern specifications, SIMPL-DIX provides an internal pattern editor with commands as illustrated in Figure 2.3. If a layer pattern is not found in the current list, SIMPL-DIX will automatically create a default pattern with the color being generated at random.

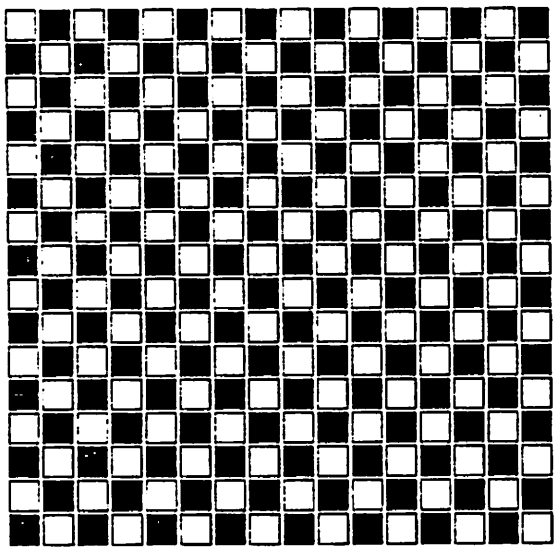


```
NAME   AIR

RGB    200 200 800

FILL   170 85 170 85 170 85 170 85
```

(A) 8x8 Bit Format



```
NAME   AIR

RGB    200 200 800

FILL   aaaa 5555 aaaa 5555 aaaa 5555 aaaa 5555
       aaaa 5555 aaaa 5555 aaaa 5555 aaaa 5555
```

(B) 16x16 Bit Format

Figure 2.2: SIMPL-DIX Pattern Specifications

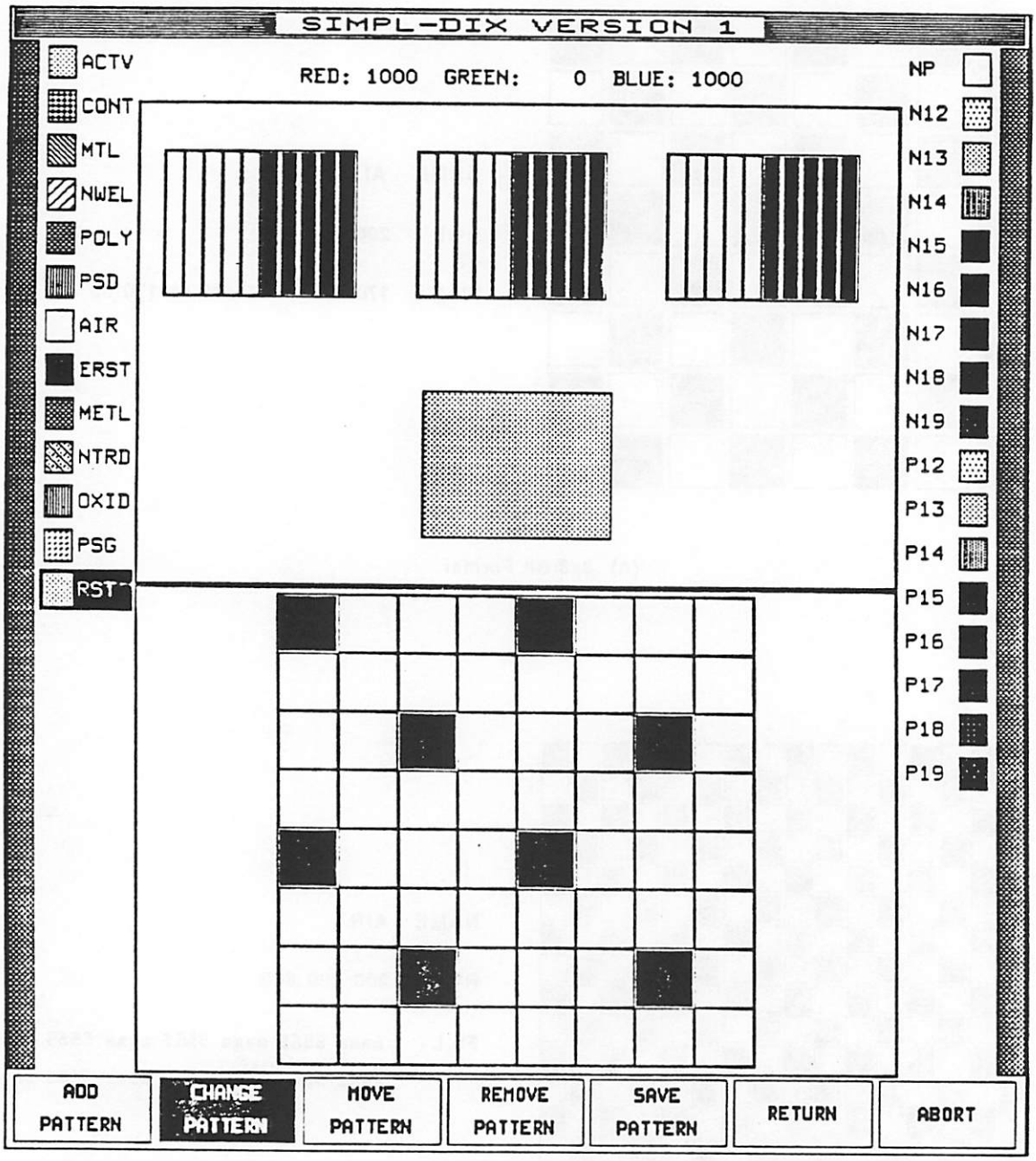


Figure 2.3: SIMPL-DIX Pattern Editor

The color and pattern information acquired from the definition file is stored in a global array *DIX_Pattern*. The basic pattern descriptor is defined in the *display.h* file as follows:

```
#define     COLOR_SIZE          117
#define     PATTERN_SIZE        85

/*
 * SIMPL-DIX pattern descriptor.
 */
typedef struct {
        char name[NAME_SIZE];       /* Pattern name. */
        short red, green, blue;     /* Color code. */
        unsigned short fill_bits[16];   /* Fill bit array. */
        Pixel color;                /* Color pixel. */
        Pixmap background;          /* Background pixmap. */
        Pixmap foreground;          /* Foreground pixmap. */
} dixPattern;

dixPattern DIX_Pattern[PATTERN_SIZE];
```

A pattern descriptor is associated with each material or mask layer defined. The *name* identifies the layer for the association; it is restricted to be up to four characters long. The *red*, *green*, and *blue* are the intensities of the three basic colors, normalized to 1000. The *fill_bits* is the array storing the stipple pattern information.

When SIMPL-DIX is invoked on a color display, a total up to COLOR_SIZE of color pixels are allocated. These *color pixels* are read/write color cells of X, and the actual number of these colors can be utilized depends on the display type. As a limited resource of X, a color pixel is released when it is no longer in use.

At start up of a SIMPL-DIX window, pixmaps for foreground and background fills are also allocated. A *pixmap* is a two-dimensional array of pixels which is used to display patterns in X. As with the color pixel, a pixmap is a limited resource of X and is freed if a pattern is no longer used.

The size of the pattern array is set to be *PATTERN_SIZE*, and with the internal filling operations performed, up to 125 patterns can actually be utilized. If the current size of 85 patterns is insufficient, this definition can be modified to allow more patterns to be specified, pro-

vided it is under the prescribed limit.

2.5. Graphics Operations

Graphics primitives such as points and lines are the base elements that compose an image [8]. Structures describing these primitives are defined in the *simpl-dix.h* file as follows:

```
/*
 * Primitive descriptors.
 */
typedef struct float_point {
        float x, y;                     /* Point coordinates. */
} floatPoint;

typedef struct float_path {
        struct float_point point;       /* Path point. */
        struct float_path *next;        /* Pointer to next path point. */
} floatPath;

typedef struct int_point {
        int x, y;                       /* Point coordinates. */
} intPoint;

typedef struct int_path {
        struct int_point point;         /* Path point. */
        struct int_path *next;          /* Pointer to next path point. */
} intPath;
```

A point is, conceptually, merely a location specified by a coordinate value; it is the building block for other primitives such as lines and polygons. Unadorned points are not often needed but do provide convenient hooks for attaching notes and referencing locations, as utilized in a number of SIMPL-DIX display routines. The doping profile display operation offers a good example of this utilization, as illustrated in Figure 2.4.

High-level application procedures also use the basic point descriptor along with its list structure *Path* to form elements such as boxes and wires. The CIF round-flash and polygon structures are included below as an illustration.

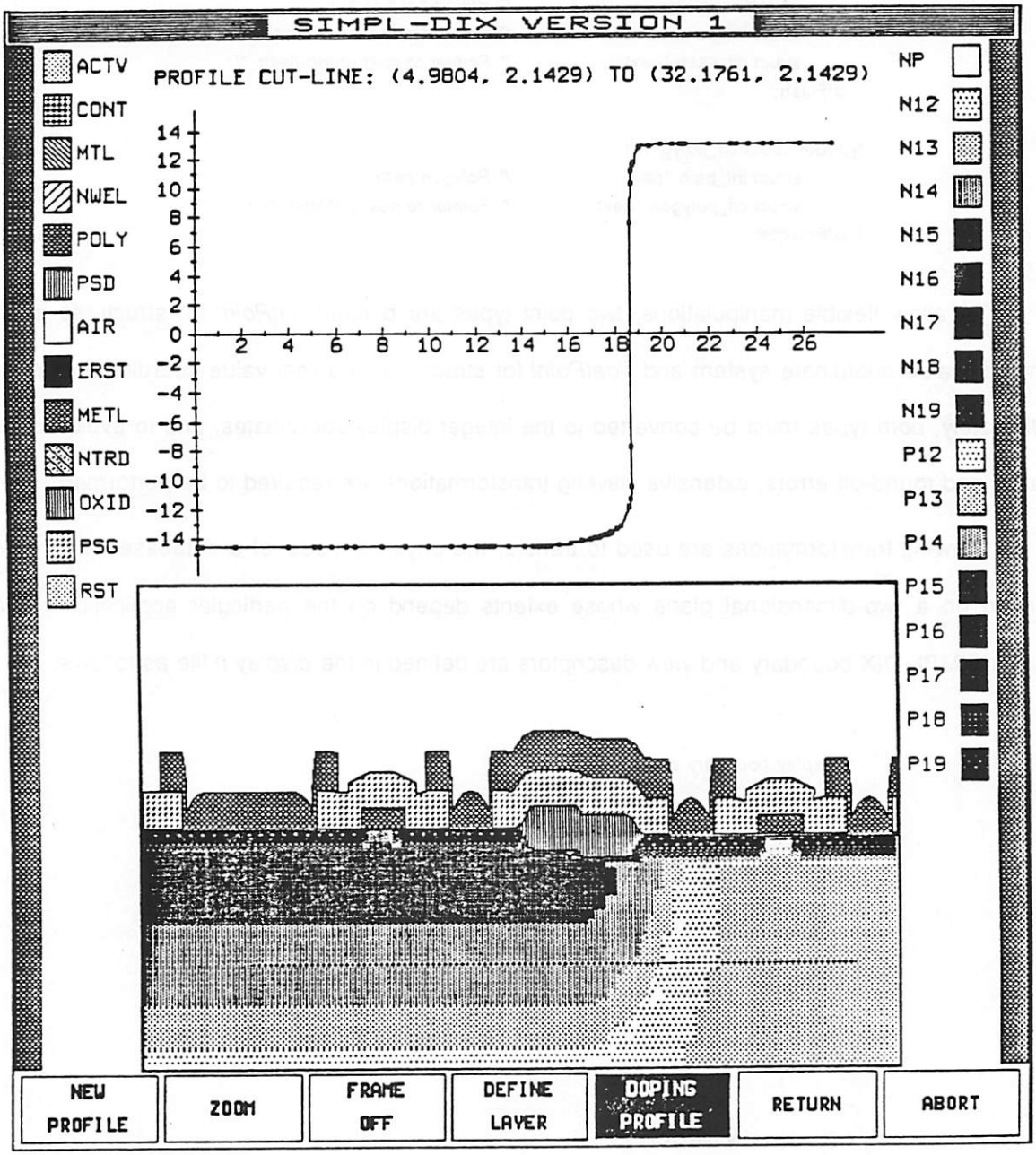


Figure 2.4: Doping Profile of a CMOS Inverter

```
typedef struct cif_flash {
        int radius;                     /* Round-flash radius. */
        struct int_point center;        /* Round-flash center point. */
        struct cif_flash *next;         /* Pointer to next round-flash. */
} cifFlash;

typedef struct cif_polygon {
        struct int_path *path;          /* Polygon path. */
        struct cif_polygon *next;       /* Pointer to next polygon. */
} cifPolygon;
```

To allow flexible manipulations, two point types are defined: *intPoint* for structures in an integer value coordinate system and *floatPoint* for structures in a real value coordinate system. Ultimately, both types must be converted to the integer display coordinates, and to avoid cumulative and round-off errors, extensive viewing transformations are required to be performed.

Viewing transformations are used to transfer the physical model of a database into a view drawn on a two-dimensional plane whose extents depend on the particular application. The basic SIMPL-DIX boundary and view descriptors are defined in the *display.h* file as follows:

```
/*
 * Display boundary descriptors.
 */
typedef struct {
        float top, bottom;              /* Vertical coordinates. */
        float left, right;              /* Horizontal coordinates. */
} floatBound;

typedef struct {
        int top, bottom;                /* Vertical coordinates. */
        int left, right;                /* Horizontal coordinates. */
} intBound;

/*
 * Display view descriptors.
 */
typedef struct {
        float x, y;                     /* Display view top-left coordinates. */
        float width, height;            /* Display view dimensions. */
} floatView;
```

```
typedef struct {
        int x, y;                       /* Display view top-left coordinates. */
        int width, height;              /* Display view dimensions. */
} intView;
```

Since the SIMPL-DIX viewports are child-windows of the main window, X conveniently provides clipping mechanism for anything drawn beyond a viewport. However, to maintain meaningful contents within a viewport, internal viewing transformations are performed. These transformations operate on the current view with respect to the boundary of an application.

Transformations for the profile data, for example, are established through procedures in the *view_control.c* file listed as follows:

```
GetProfileViewX(viewport, profile_view, profile_x)
        dixViewport viewport;
        floatView profile_view;
        float profile_x;

GetProfileViewY(viewport, profile_view, profile_y)
        dixViewport viewport;
        floatView profile_view;
        float profile_y;

float
GetProfileX(viewport, profile_view, viewport_x)
        dixViewport viewport;
        floatView profile_view;
        int viewport_x;

float
GetProfileY(viewport, profile_view, viewport_y)
        dixViewport viewport;
        floatView profile_view;
        int viewport_y;
```

GetProfileViewX() and *GetProfileViewY()* transform the prescribed *profile_x* and *profile_y* to those of the viewport coordinates according to the specified *profile_view*. The contents in *profile_view* are referenced to a global boundary *Profile_Bound* for the profile system. Conversely, *GetProfileX()* and *GetProfileY()* transform the specified *viewport_x* and *viewport_y* to those of the profile real value coordinates. The viewing transformation executed by the command *Zoom* is illustrated in Figure 2.5.

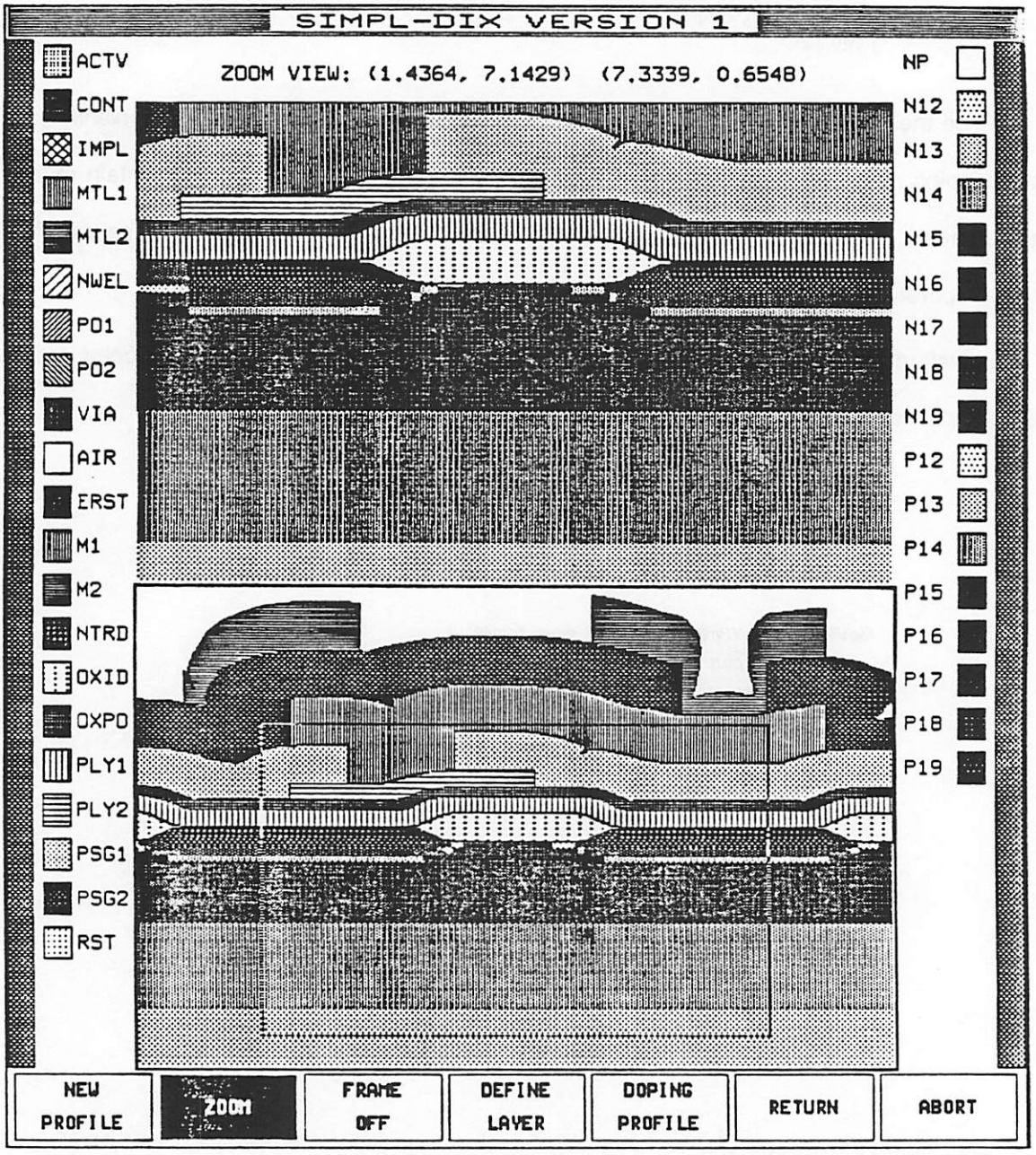


Figure 2.5: Zooming Operation with a CDRAM Profile

2.6. Text Operations

To enhance text operations for the X window system, a number of SIMPL-DIX display procedures are developed. These routines generally adopt definitions in the *display.h* file as listed below:

```
/*
 * Text operation definitions.
 */
#define     NO_CLIP          0
#define     CLIP_LEFT        1
#define     CLIP_RIGHT       2
```

NO_CLIP specifies that no clipping operation is to be performed when text is displayed. *CLIP_LEFT* specifies that left-end characters are clipped in case a text string cannot be completely displayed; *CLIP_RIGHT* specifies that right-end characters are dropped off when part of the text cannot appear.

Internal SIMPL-DIX textual operation procedures in the file *graphics_control.c* are describeed in details as follows:

```
GetTextWidth(text, size, font_info)
        char *text;
        int size;
        FontInfo *font_info;
```

The procedure *GetTextWidth()* calculates the width in pixels of the specified string using the specified X font structure *font_info*. Only the specified number *size* of characters are used in the calculation. Although a number of X routines provide similar operations, utilizations of these library procedures are time consuming and often cause errors. Thus, this routine is implemented to provide a precise control for the placement of SIMPL-DIX text information; it allows fonts with non-fixed width to be used. The list of available fonts for X can be found in the directory */usr/new/lib/X/font*.

```
PrintCenteredText(viewport, x, y, width, height, text, font_info,
    foreground, background, text_func)
        dixViewport viewport;
        int x, y, width, height;
        char *text;
        FontInfo *font_info;
        Pixel foreground, background;
        int text_func;
```

The procedure *PrintCenteredText()* is the basic display routine to write text in the center of a box. Point (*x*, *y*) is the upper-left corner of the bounding box. The specified *foreground* and *background* color pixels are used to display the text. The flag *text_func* uses constants *NO_CLIP*, *CLIP_LEFT*, and *CLIP_RIGHT* to determine how the text should appear in case the string cannot be fitted in the box.

```
PrintText(viewport, x, y, width, height, text, font_info,
    foreground, background, text_func)
        dixViewport viewport;
        int x, y, width, height;
        char *text;
        FontInfo *font_info;
        Pixel foreground, background;
        int text_func;
```

Similar to *PrintCenteredText()*, the procedure *PrintText()* also writes text in a box, but the string is left-adjusted for display. Point (*x*, *y*) is the upper-left corner of the box; the box is used as a bounding box for controlling the appearance of the string *text*. All SIMPL-DIX display routines use the above procedures to perform textual operations.

Chapter 3

Application Interface with SIMPL-2

3.1. Introduction

As a high-level design interface tool, SIMPL-DIX relies on external programs such as SIMPL-2 to perform basic simulation operations. For an application with SIMPL-DIX, information supplied by the user is organized and sent to the simulators; data generated by the external tools are collected and utilized for the user. Through proper control of the application interfaces, integration of simulations can be achieved for an optimum design environment.

In an application interface, the burden of manipulating the external tool should be placed on the high-level controller. Ideally, a normal user of SIMPL-DIX should not be aware of the existence of the underlying simulators; the application interface should make the composite design tool appeared as a single entity. In addition, the simulation operations should not be hindered by the fact that several programs are running together; simulators should be able to attain their normal operating conditions despite that they are actually controlled by a high-level procedure. Hence, an important task for SIMPL-DIX to achieve is the establishment of smooth and efficient communication links with other programs.

The design of the interface between SIMPL-DIX and SIMPL-2 is discussed in section 3.2; the model described there is intended to serve as a prototype for other simulator interface. Methodology for future application interface is discussed in section 3.3.

3.2. Interface with SIMPL-2

SIMPL-2 is a CAD tool which simulates the topography of an integrated circuit using two-dimensional process models. It accepts either a process description file or a sequence of commands to generate the cross-sectional profile along an arbitrary cut-line drawn on the layout. For an application with SIMPL-DIX, the profile data produced by SIMPL-2 are used as the seed

database for simulations.

The interface between SIMPL-DIX and SIMPL-2 is accomplished through the use of the interprocess communication (IPC) facilities in the Berkeley UNIX 4.3BSD release [9]. The procedure *SIMPLConnect()* in the source file *simpl_interface.c* is included in Figure 3.1 as an illustration.

In IPC, a *socket* is a transient object representing an endpoint of communication; it is the portal through which data can be sent or received. In addition, a *connected* socket is a socket which has the property that any data written on it will always be sent to a certain address, and any data received on it will have come from the same address.

With each invocation of SIMPL-DIX, a pair of connected sockets, *FD[0]* and *FD[1]*, is created by calling the system routine *socketpair()*. The arguments *AF_UNIX* and *SOCK_STREAM* in the system call specify the domain and the style of communication, respectively. Both constants are defined in *<sys/socket.h>*, which in turn requires the file *<sys/types.h>* for some of its definitions.

After the creation of the socket pair, the process running SIMPL-DIX is split into two processes through *fork()* [10]. While the *parent* process continues to run SIMPL-DIX, the *child* process begins redirecting its standard input and output file descriptors to one end of the sockets, *FD[0]*, and starts the execution of SIMPL-2 in non-graphics mode using *execl()*. Since SIMPL-2 reads from *stdin* and writes to *stdout*, this approach ensures a two-way stream communication channel between the two processes is established.

The global variable *SIMPL_Path* specifies the location of SIMPL-2 on the system; its default setting is defined in the *default.h* file as */cad/bin/simpl-2*. As with many other global variables, its default value can be overridden by the user through a specification in the *.Xdefaults* file. Appendix A includes a summary of available default specifications.

```
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>

extern char *SIMPL_Path;

static short SIMPL_Status;
static int SIMPL_Pid;
static int FD[2];

/*****************************************************************************
 * SIMPLConnect:
 *      This routine establishes connection between SIMPL-DIX and SIMPL-2.
 *
 * RETURN VALUE:
 *      SUCCESS or FAIL
 *****************************************************************************/

SIMPLConnect()
{
        int SIMPLErrorHandler();
        int SIMPLQuit();

        if (SIMPL_Status == SET) {
                return(SIMPLReset());
        }
        if (socketpair(AF_UNIX, SOCK_STREAM, 0, FD) < 0) {
                WriteError("SIMPL ERROR : Cannot create SIMPL-2 socket");
                return(FAIL);
        }
        if ((SIMPL_Pid = fork()) < 0) {
                WriteError("SIMPL ERROR : Cannot create SIMPL-2 process");
                return(FAIL);
        }
        if (SIMPL_Pid == 0) {
                /* Process to run SIMPL-2. */
                close(0);
                dup2(FD[0], 0);
                close(1);
                dup2(FD[0], 1);
                close(2);
                dup2(FD[0], 2);
                execl(SIMPL_Path, "simpl-2", "-n", 0);
                exit(ERROR);
        } else {
                /* Process to run SIMPL-DIX. */
                signal(SIGALRM, SIMPLQuit);
                signal(SIGCHLD, SIMPLErrorHandler);
                SIMPL_Status = SET;
                return(SIMPLSet());
        }
}
```

Figure 3.1: Procedure to Connect SIMPL-2

Now, whenever the user enters a command needs to be executed by SIMPL-2, the parent process running SIMPL-DIX writes the information to the socket *FD[1]*, which will be received by the child process through *FD[0]*. The information supplied by the user is stored temporarily in the global buffer *Input_Buffer*. This information is obtained through internal textual operations such as *SIMPLGetAnswer()* in either *SIMPLRunBatch()* or *SIMPLRunInput()*. The output action for the interface with SIMPL-2 is illustrated by the procedure *SIMPLWrite()* in Figure 3.2

After SIMPL-DIX has sent a command to SIMPL-2, it waits for the child process to request either additional information or a next command. The request made by SIMPL-2 will be received through the socket *FD[1]*, as illustrated by the procedure *SIMPLRead()* in Figure 3.3.

While SIMPL-DIX is waiting for SIMPL-2 to make the next request, the parent process continues to interact with the X server. For an interval of one second, SIMPL-DIX listens to the channel for any incoming data through *select()*. If nothing appropriate has appeared, it precedes with display operations and listens to the channel again. This process is repeated until the prescribe time *TIME_OUT* (10 minutes) has expired, with which the routine *SIMPLQuit()* will be executed to terminate the connection with SIMPL-2.

```
/*****************************************************************************
 * SIMPLWrite:
 *      Add a new-line character to Input_Buffer and send the data to SIMPL-2.
 *****************************************************************************/

SIMPLWrite()
{
        int n_bytes, n_write;

        n_bytes = strlen(Input_Buffer);
        Input_Buffer[n_bytes++] = CR;
        Input_Buffer[n_bytes] = EOS;

        n_write = write(FD[1], Input_Buffer, n_bytes);
        if (n_write < 0) {
                WriteError("SIMPL ERROR : Cannot write to SIMPL-2");
                SIMPLQuit();
        }
}
```

Figure 3.2: Procedure to Write Data to SIMPL-2

```
/*****************************************************************************
 * SIMPLRead:
 *      Read stream of data from the output file descriptor of SIMPL-2.
 *      If an error occurred (either in read or time out), SIMPLQuit is executed.
 *
 * RETURN VALUE:
 *      The number of bytes read into Input_Buffer.
 *      If time out before anything is read, -1 is returned.
 *****************************************************************************/

SIMPLRead()
{
        struct timeval wait_time;
        fd_set read_set;
        int n_read;
        int status;

        wait_time.tv_sec = 1;
        wait_time.tv_usec = 0;
        alarm(TIME_OUT);

        loop {
                while (XPending() > 0) {
                        /* Process all X events. */
                        if (GetEvent(AllEvents) == ExposeWindow) {
                                ...
                                break;
                        }
                }
                FD_ZERO(&read_set);
                FD_SET(FD[1], &read_set);
                status = select(FD_SETSIZE, FD_MASK(&read_set),
                    FD_MASK((fd_set *) 0), FD_MASK((fd_set *) 0), &wait_time);
                if (status > 0) {
                        /* SIMPL-2 is ready to be read. */
                        break;
                } else if (status < 0) {
                        WriteError("SIMPL ERROR : Connection error");
                        return(-1);
                }
        }
        alarm(0);
        n_read = read(FD[1], Input_Buffer, BUFFER_SIZE);
        if (n_read < 0) {
                WriteError("SIMPL ERROR : Cannot read from SIMPL-2");
                SIMPLQuit();
                return(-1);
        } else if (n_read < BUFFER_SIZE) {
                Input_Buffer[n_read] = EOS;
                return(n_read);
        }
}
```

Figure 3.3: Procedure to Read Data from SIMPL-2

The macro *FD_MASK()* in the system call *select()* is added for compatibilities between 4.3BSD and ULTRIX-32. Since ULTRIX-32 does not provide definitions such as *FD_SETSIZE*, *FD_SET()*, and *FD_ZERO()* needed for the *select()* operation, these macros are explicitly included in the *simpl-dix.h* header file. Thus, when installing SIMPL-DIX on a system, the macro *SYSTEM* in *Makefile* needs to be set as *BSD* for 4.3BSD or as *ULTRIX* for ULTRIX-32.

Upon termination of SIMPL-2 either at time-out or by an explicit command from the user, the routine *SIMPLErrorHandler()* checks the exit status of SIMPL-2; it outputs error messages generated by SIMPL-2 in case a problem occurred.

On the other hand, after a command has been successfully executed, the resulting data generated by SIMPL-2 are saved in a file, which is read in by SIMPL-DIX. The data acquired are stored in two formats: a grid data type for the substrate structure and a linked polygon list for material layers.

The grid data structure is defined in files *dix_init.c* and *simpl.h* as follows:

```
/*
 * Profile substrate information.
 */
#define     GRID_X_SIZE          150
#define     GRID_Y_SIZE          100

float SIMPL_Doping[GRID_X_SIZE][GRID_Y_SIZE];
float SIMPL_GridX[GRID_X_SIZE];
float SIMPL_GridY[GRID_Y_SIZE];
```

The profile polygon descriptor is defined in the *simpl.h* file as follows:

```
/*
 * Profile polygon descriptor.
 */
typedef struct simpl_polygon {
        char name[NAME_SIZE];       /* Polygon layer name. */
        struct float_path *path;    /* Polygon path. */
        struct simpl_polygon *next; /* Pointer to next layer. */
} simplPolygon;

simplPolygon *SIMPL_PolygonRt;
```

This structure contains the minimum information needed to fully describe a material layer; it allows transformation to other database be easily established. High-level display routines are built to utilize this structure for drawing the cross-sectional profile.

3.3. Future Interface with External Simulators

In general, three major issues are involved in a design of an application interface. The *communication channel establishment* determines the means of data transmission between two programs. The *protocol specifications* define the rules or data formats for communication between two applications. The *post-transmission operations* control the acquired data for high-level display and other utilizations.

As with SIMPL-2, most simulators accept commands either interactively or in batch mode. With the interactive systems, the definite choice for the communication channel is to use IPC facilities such as sockets. For systems running in batch mode, IPC should also be utilized to eliminate the time required to set up a secondary process.

The major advantage gained with the IPC scheme is that it allows SIMPL-DIX to have flexible control over external simulators. Internally, SIMPL-DIX only needs to know when to expect a request from the simulator and when to get an input from the user. It does not have to check the format or validity of the input. Since different simulators have different command or data requirements, the verification tasks should be performed by the external simulators. Thus, if an invalid input has been sent, it should be the responsibility of the external simulator to inform SIMPL-DIX the error condition and to request a new input.

In addition, by maintaining minimum restrictions on the formats of inputs, external simulators can be developed independently of one another. This is a desirable feature which allows intergration of different simulators for various applications. The only thing needs to be agreed between SIMPL-DIX and an external simulator is the specification of a transmission protocol. Within the protocol, for example, SIMPL-DIX only needs to know how to decide if the simulator

is ready for an input or an output.

However; in order to establish a smooth communication link, a requirement for simulators using standard I/O operations such as *printf()* and *scanf()* is that either a new-line character has to be sent or the file buffers have to be flushed, which can be accomplished using *fflush()* system call for the simulator part. An alternative approach is for SIMPL-DIX to use library routines such as *fgets()* and *fputs()*, but when using sockets, these operations can only manipulate one byte at a time, which may not be efficient if a large amount of data transfer is required.

In addition, for program such as SIMPL-2 which does not read and write the simulation data directly, file transferring of data is required. This process might slow down the overall operation, but the problem can be solved in the future if a common database shared by different simulators can be established.

Finally, a transformation of the data sent by other simulators to the internal SIMPL-DIX structure is required; special display routines might even needed to be built if the data do not fit into the existing framework. Thus, a standard data description such as the proposing profile interchange format (PIF) is desperately needed for an ideal integrated CAD environment [11].

Chapter 4
Mask Operations with HUNCH

4.1. Introduction

Each process used to fabricate integrated circuits is characterized by a set of layout rules that must be observed when using the process. These rules which specify geometric constraints for the mask artwork are established to eliminate sensitivity of IC topology to process instabilities. As a communication link between circuit designers and process engineers, these layout rules represent the best possible compromise between device performance and manufacturing yield.

While actual layout rules must be based on experience gained on fabrication lines, the spiraling cost of process operations has made it desirable to optimize design parameters prior to trial fabrications. The growing circuit and process complexity unfortunately has become so prohibitive that manual analysis of the interrelationships between layout levels and process sequences is quite impracticable, and new approaches in a CAD environment for the formulation of layout rules are needed.

HUNCH is an internal tool of SIMPL-DIX developed to alleviate the problem in perceiving the complex process-design relationship for IC fabrication. Using sequences of algebraic operations, layers prescribing problematic regions are created from the original masks, and through process simulations, potential topographical errors associated with the derived layers can be espied.

4.2. HUNCH Layer Description

An important concept applied in the analysis and synthesis of layout rules is the definition of abstract layers [12-15]. Internally, HUNCH treats each mask layer as a collection of geometric entities and adopts an algebra for manipulating and combining the segregated consti-

tuents. The algebra defined includes operations such as geometrical operations, logical operations, and topological operations, as illustrated in Figure 4.1. Using this generic algebra, abstract layers are created from the original masks, and these layers form the basis for analyzing layout rules.

HUNCH accepts layer descriptions either interactively or from a textual file. The syntax for the layer description is as follows:

< LAYER NAME > : < ALGEBRAIC SPECIFICATION >;

Each HUNCH layer is identified by a name, restricted to be up to four characters long. Associated with each layer is an algebraic specification prescribing the sequence of operations to be performed. Available operations and their formats are summarized in Table 4.1. Salient features of selected operations are described below.

When input is read from a HUNCH definition file, comments can be included to ascribe physical properties associated with each derived layer. In each description, text following the separator <*> is treated as comment. Additional comments preserving diagnostic messages are generated by HUNCH whenever syntax errors occur.

In the algebraic specification, the *Layer* can be either a CIF mask layer, a HUNCH abstract layer, or a compound expression involving both types. Conventional infix notation is adopted, with unary operators preceding their arguments and binary operators in-between their operands. The order of operation is left-to-right, but parentheses can be used to group subexpressions.

HUNCH operations are loosely classified as follows: *geometrical* operations to perform geometrical transformations on figures of the specified layers, *logical* operations to produce logical combinations of regions for the specified layers, and *topological* operations to recognize relationships among entities of the specified layers. A group operation is also provided to join a set of layers within a single definition; this function is equivalent to a combination of logical addition operations.

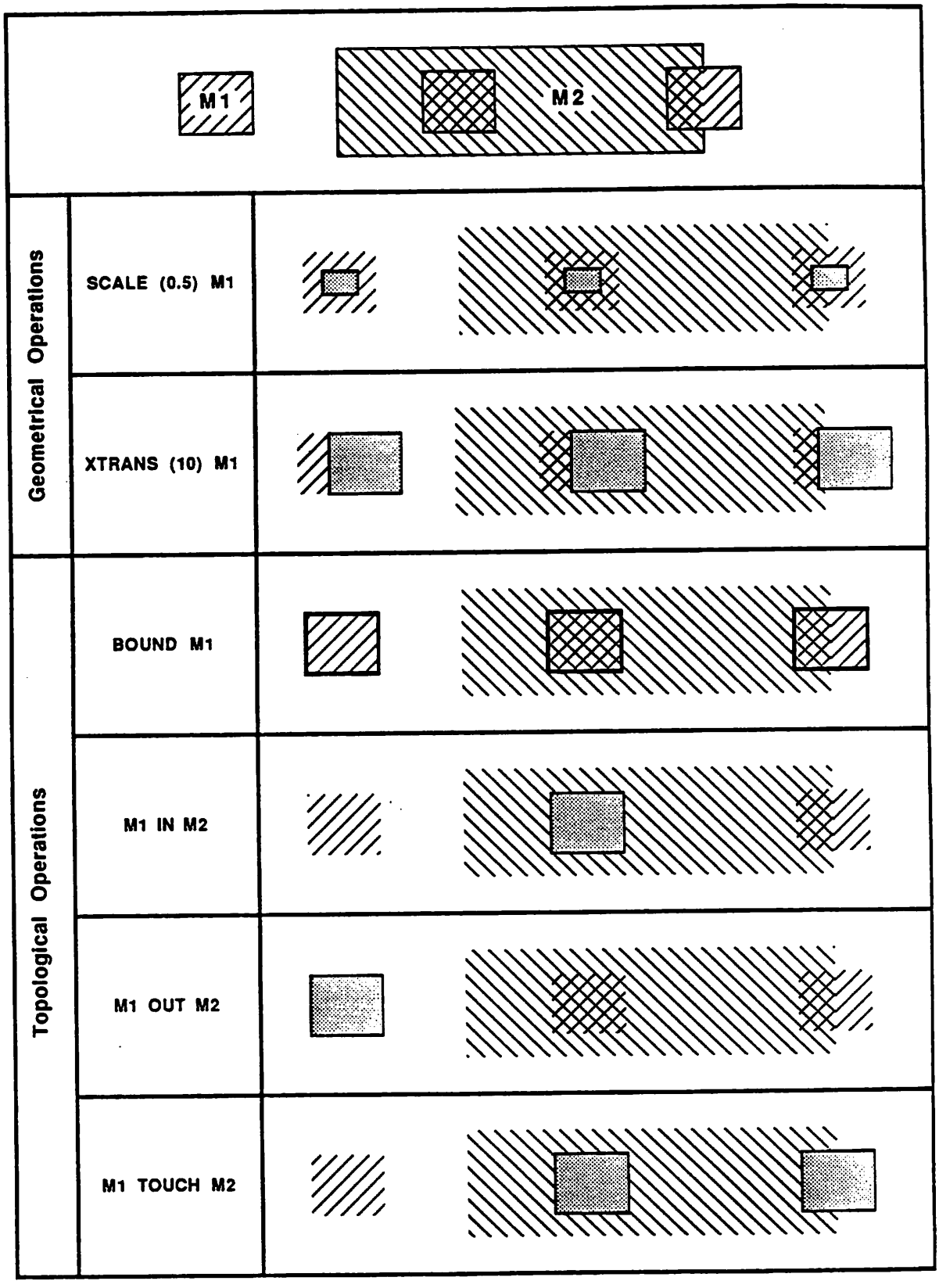


Figure 4.1: Mask Operations

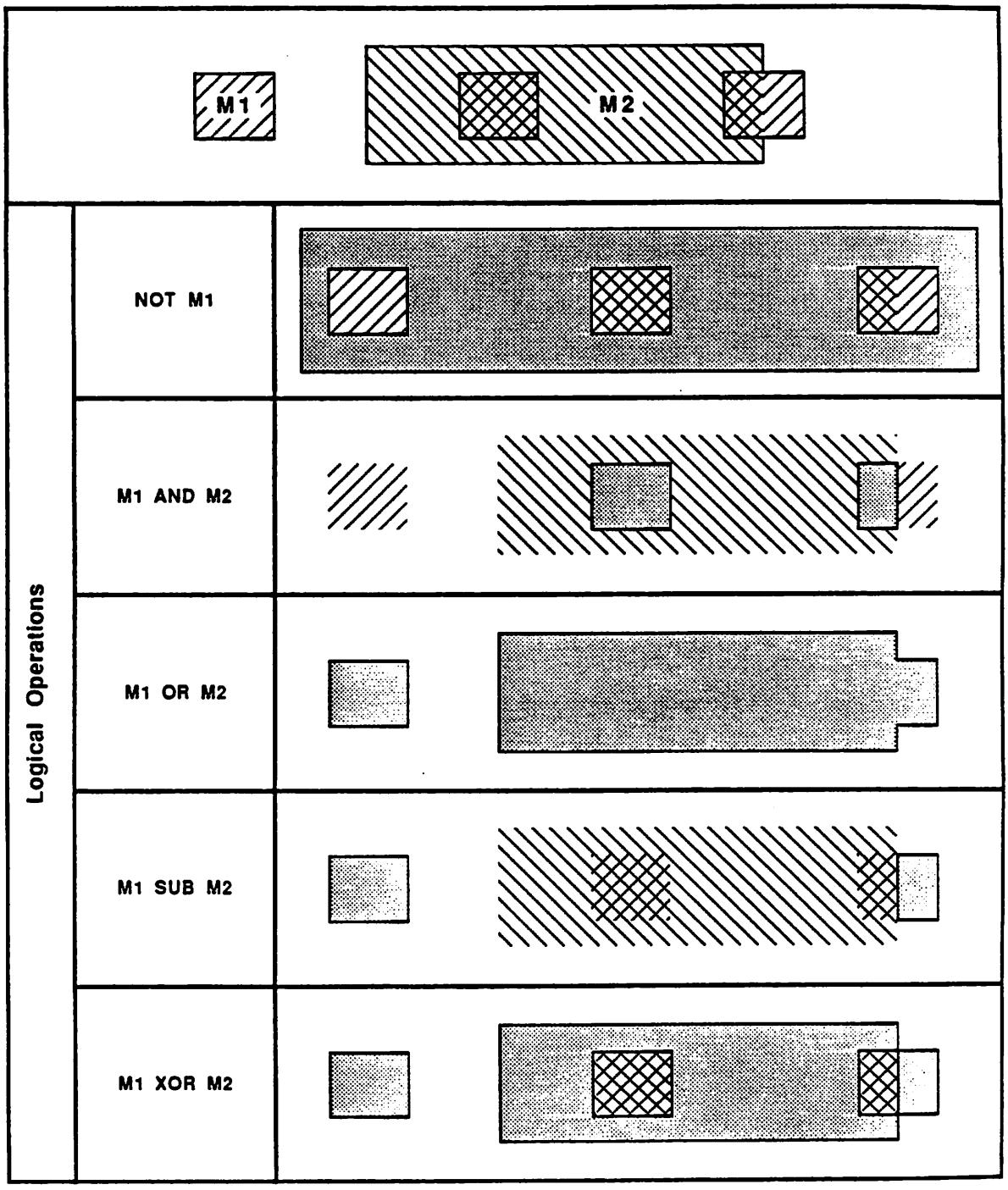


Figure 4.1: (Continued)

OPERATION	FORMAT
Group Operation	SET (*Layer, Layer, ...*)
Geometrical Bloat Operation	BLOAT (*IntFactor*) *Layer*
Geometrical Scale Operation	SCALE (*FloatFactor*) *Layer*
Geometrical Translation Operation	XTRANS (*IntFactor*) *Layer*
Geometrical Translation Operation	YTRANS (*IntFactor*) *Layer*
Logical Complement Operation	NOT *Layer*
Logical Addition Operation	*Layer* OR *Layer*
Logical Intersection Operation	*Layer* AND *Layer*
Logical Subtraction Operation	*Layer* SUB *Layer*
Logical Exclusive-or Operation	*Layer* XOR *Layer*
Logical Inverse-and Operation	*Layer* NAND *Layer*
Logical Inverse-or Operation	*Layer* NOR *Layer*
Topological Boundary Operation	BOUND *Layer*
Topological Inside Operation	*Layer* IN *Layer*
Topological Outside Operation	*Layer* OUT *Layer*
Topological Touch Operation	*Layer* TOUCH *Layer*

Table 4-1: HUNCH Operations

For all geometrical operations defined, a numerical value is required to be the first argument. Two types of scaling operations are provided; the difference between these two functions is that BLOAT grows or shrinks the specified layer by the specified amount, whereas SCALE multiplies dimensions of the specified layer by the specified factor. Translation operations XTRANS and YTRANS are applied to shift coordinates of entities by the specified amount along the prescribed direction. Together, these geometrical operations provide simple means for

modeling bias and misalignment effects.

Basic logical operations are consisted of the complement NOT, addition OR, and intersection AND functions. In a layer description, the binary operations AND and OR are used to manipulate areas between the two specified layer operands; the complement function NOT is applied to obtain the negative of regions defined by the the specified layer confined in the global layout boundary. Other operations such as SUB and XOR are derived from combinations of these primitive logical functions.

Finally, to recognize rules associated with specific devices on an IC, topological operations are used to determine the relationship of one entity to another. In the CMOS technology, for example, guard bands usually surround a transistor to prevent field inversion of the silicon surface [16]. Violations of such rules can be perceived through layers derived with operations such as IN or OUT. In particular, the result of the operation IN is the geometric entities of the first layer that are strictly inside regions defined by the second layer, whereas the product of OUT is the geometries of the first layer strictly outside the second layer. The output of TOUCH is the geometries of the first layer either inside or touching regions of the second layer. Unlike logical functions, these topological operations preserve connectivity within each geometric entity.

As a simple illustration of HUNCH mask operations, Figure 4.2 shows the topography of a CMOS inverter with the metal layer forming an undesirable corner-cut. Regions to be verified can be expressed as:

```
*****************************
*  Metal Corner Cut.  *
*****************************

CUT : MTL AND (BLOAT (70) POLY);
```

Figure 4.3 illustrates the step coverage problem for a CDRAM structure, with HUNCH layers as highlighted.

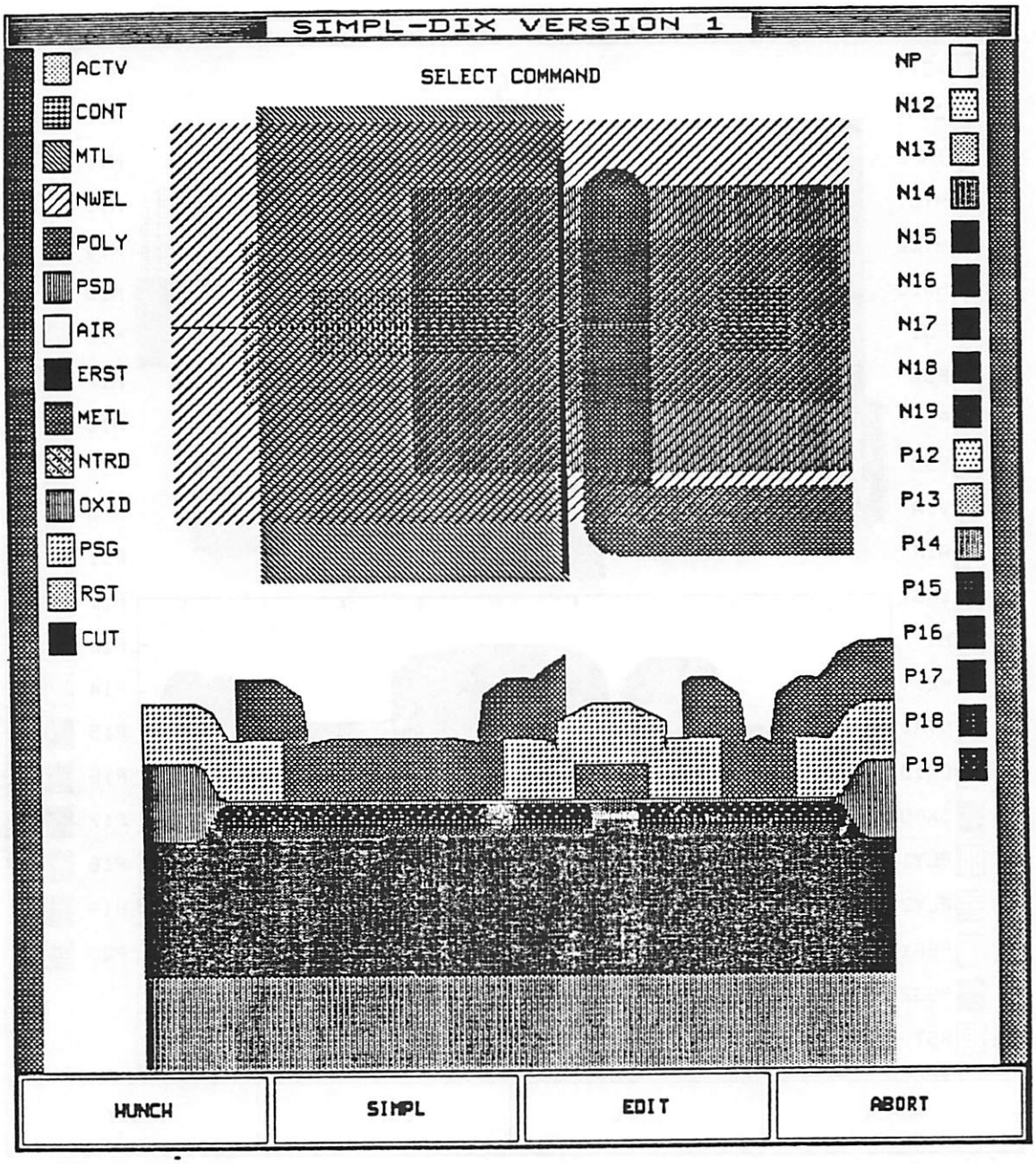


Figure 4.2: CMOS Metal Corner Cut

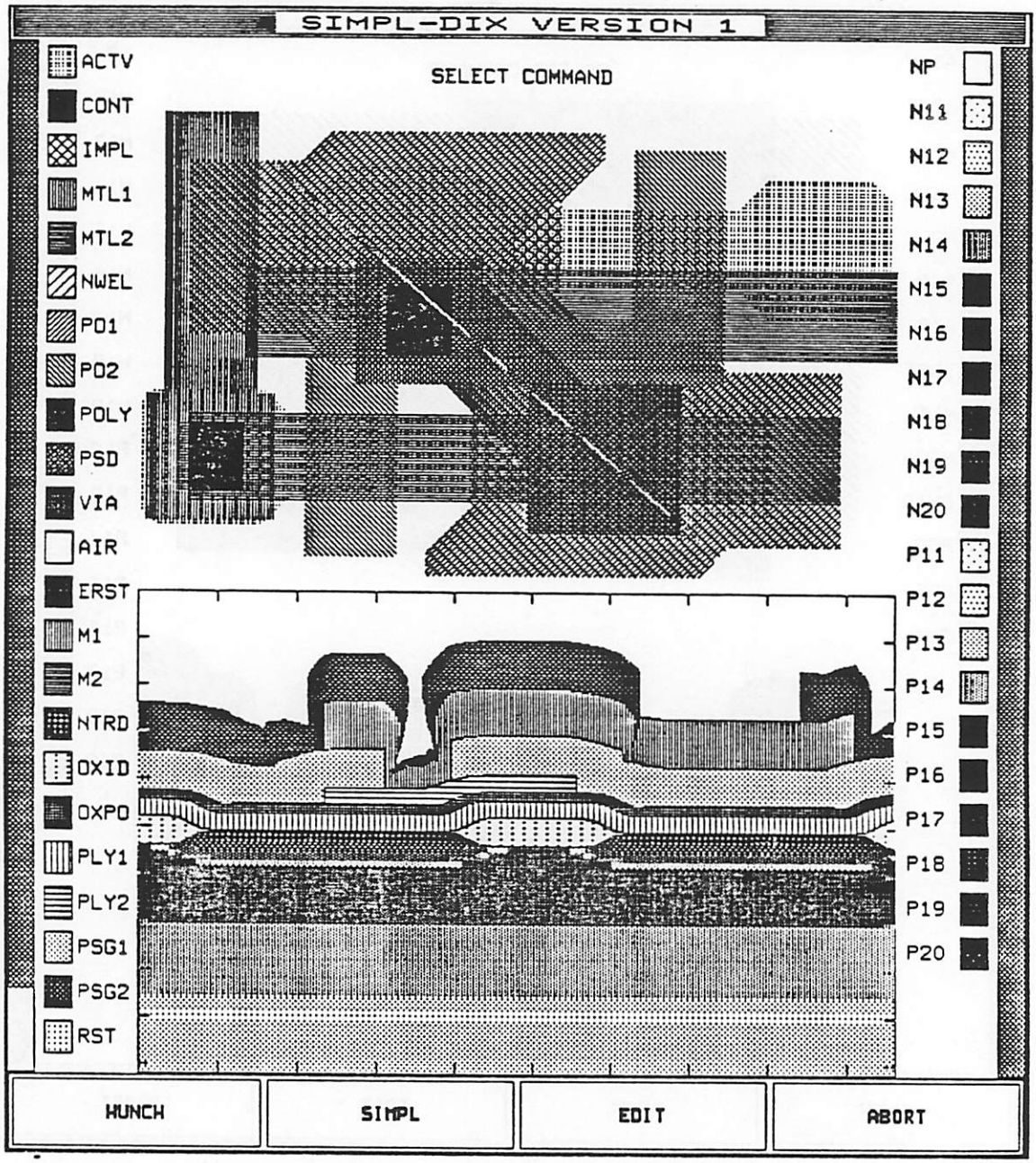


Figure 4.3: CDRAM Step Coverage problem

4.3. HUNCH Layer Structure

This section describes the various procedures and descriptors used to construct the abstract HUNCH layer. Data structures, macros, and symbols used by HUNCH are defined in the *hunch.h* file.

The basic HUNCH layer descriptor is defined as follows:

```
/*
 * HUNCH layer descriptor.
 */
typedef struct hunch_layer {
        short active;                   /* Activation flag. */
        char name[NAME_SIZE];           /* Layer name. */
        struct hunch_op *op_rt;         /* Pointer to operation tree. */
        struct hunch_com *com_rt;       /* Pointer to comment list. */
        struct hunch_geo *geo_rt;       /* Pointer to geometry list. */
        struct hunch_cut *cut_rt;       /* Pointer to cut-edge list. */
        struct hunch_layer *next;       /* Pointer to next layer. */
} hunchLayer;
```

A layer descriptor is associated with each HUNCH layer as a node in a singly linked list. The global variable *HUNCH_LayerRt* is the pointer to the head node of the layer list.

For each HUNCH layer defined, the activation flag *active* controls the layer as being visible or invisible. Normally, the flag takes on the value of *TRUE* or *FALSE*, which can be toggled by the command *Set_HUNCH_Mode*. When syntax errors for a layer description occur, the flag assumes the value of *ERROR*, and diagnostic messages are generated and stored as comments to be associated with the layer structure. Only visible layers are saved when the command *Save_HUNCH_Layer* is invoked.

The field *op_rt* of the layer structure is the pointer to the root of the operation tree constructed by the HUNCH parser module. Routines in *hunch_parser.c* are executed to scan the layer description for tokens such as arguments or operators. Upon detection of a valid token, action procedures in *hunch_action.c* are invoked to create proper a node for the operation tree.

Two types of descriptors are used as the building blocks for the operation tree:

```
/*
 * HUNCH operator descriptor.
 */
typedef struct hunch_op {
        int code;                       /* Operator code. */
        float scale;                    /* Scale factor. */
        struct hunch_arg *arg1;         /* Pointer to first argument. */
        struct hunch_arg *arg2;         /* Pointer to second argument. */
        struct hunch_arg *arg_bptr;     /* Backward argument pointer. */
} hunchOp;

/*
 * HUNCH argument descriptor.
 */
typedef struct hunch_arg {
        char name[NAME_SIZE];           /* Argument name. */
        struct hunch_op *op_fptr;       /* Forward operator pointer. */
        struct hunch_op *op_bptr;       /* Backward operator pointer. */
        struct hunch_arg *next;         /* Pointer to next argument. */
} hunchArg;
```

A node of the operation tree consists of an operator descriptor followed by one or two argument lists. Each operator descriptor is doubly linked to an argument descriptor through *arg_bptr*, and each argument descriptor is doubly linked to an operator descriptor through *op_bptr*. These linkages provide essential means to generate structures with subexpressions, as utilized by the action procedures *HUNCHDoArg()* and *HUNCHDoMoveBack()*. Operation tree for the CMOS metal cut example is depicted in Figure 4.4.

Within the operator structure, the field *code* identifies the type of the mask operation to be performed, and the field *scale* stores the scaling factor effective for the linked layer operands. For an unary operator, only *arg1* list is allocated; for a binary operator, both *arg1* and *arg2* lists are utilized.

Two types of arguments are considered, simple and compound. For a simple argument, *name* contains the identifier for the layer operand, which can be either a current CIF layer or a previously defined HUNCH layer. For a compound argument, the field *name* is untenanted, but a subtree is linked through the forward pointer *op_fptr*. In case the group operation SET is applied, subsequent layer arguments are joined together with the current operator.

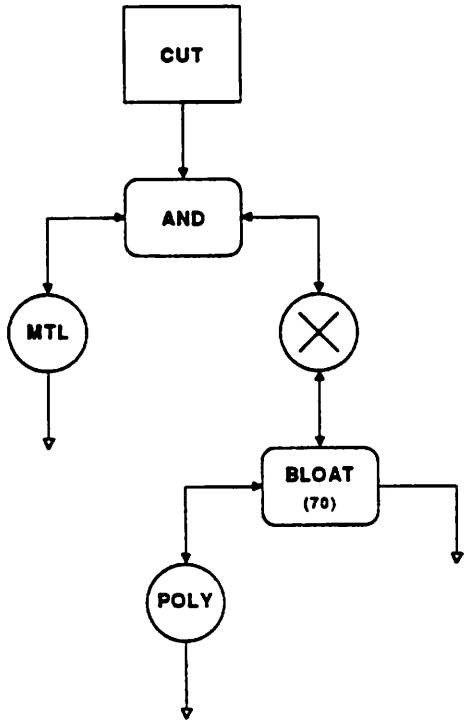


Figure 4.4: HUNCH Operation Tree

The field *com_rt* of the layer structure is the pointer to the head node of the singly linked list storing comments. The descriptor is defined as follows:

```
/*
 * HUNCH comment descriptor.
 */
typedef struct hunch_com {
        char *text;                     /* Comment text. */
        struct hunch_com *next;         /* Pointer to next comment. */
} hunchCom;
```

Each comment read in from the definition file is stored in a null-terminated string *text*. When syntax errors for a layer description occur, additional comments preserving diagnostic messages are generated by the action procedure *HUNCHDoError()*. Descriptions of syntax errors are summarized in Table 4.2.

MESSAGE	DESCRIPTION
Invalid Argument	Argument is parsed while argument field has been filled.
Invalid Operator	Binary operator is parsed while argument field has not been filled.
	Unary operator is parsed while argument field has been filled.
Invalid Opening Parenthesis	Binary operator with opening parenthesis is parsed.
Invalid Closing Parenthesis	Argument with closing parenthesis is parsed while operator field has not been filled.
	Operator with closing parenthesis is parsed.
Unmatched Parenthesis	Unmatched parenthesis is parsed.

Table 4-2: HUNCH Syntax Error Descriptions

The field *geo_rt* of the layer structure is the pointer to the head node of the linked list storing geometric data. Structures used are:

```
/*
 * HUNCH geometry descriptor.
 */
typedef struct hunch_geo {
        struct hunch_box *box_rt;       /* Pointer to box list. */
        struct hunch_geo *next;         /* Pointer to next geometry. */
} hunchGeo;

typedef struct hunch_box {
        int left, right;                /* Horizontal coordinates. */
        int top, bottom;                /* Vertical coordinates. */
        struct hunch_box *next;         /* Pointer to next box. */
} hunchBox;
```

A geometry descriptor is associated with each geometric entity defined or generated in a HUNCH layer. Primitive such as box, wire, polygon, or round-flash is represented indistinctly by a list of rectilinear box structures. Global layout coordinate system is used for dimensional

context of each box structure.

Inherently, HUNCH operations relies on the *FANG* procedures developed by the Berkeley CAD group [17]. This library package can only perform logical operations with Manhattan geometries; consequently, the HUNCH geometry is limited to the rectilinear box structures. For non-Manhattan geometry, HUNCH uses the minimum bounding box of the geometry in the mask operations. Since FANG has its internal geometry description, the routines *HUNCHToFANG()* and *FANGToHUNCH()* in the file *hunch_operation.c* are provided for transformations between these two distinct structures.

Finally, profile information is coupled with the HUNCH layer structure through a list of cut-edge descriptors, defined as follows:

```
/*
 * HUNCH cut-edge descriptor.
 */
typedef struct hunch_cut {
        float left, right;              /* Cut-edge pair. */
        struct hunch_cut *next;         /* Pointer to next cut-edge. */
} hunchCut;
```

The cut-edge descriptor represents the layer edge incised by the layout cut-line. For each activated layer, the list of cut-edges is evaluated, and regions of the cross-sectional profile bounded by the corresponding pairs *left*, *right* are highlighted.

Chapter 5
Future Extensions

5.1. Introduction

The initial structure of SIMPL-DIX has been established; however, expansions and improvements are needed to make this design tool truly an integrated CAD system. Future work should be concentrated on interfaces with external process and device simulators. Construction of additional modules for mask operations should also be required. Finally, for an improved user interface, some emendations on the display structure should be performed.

5.2. Display Interface

A number of facilities for the X window system have not been utilized in the initial version of SIMPL-DIX; in the future, they should be incorporated for an improved user interface. In particular, a toolkit such as *XMenu* which provides a *deck of cards* menu system should be adopted. With this type of menus, excess area for the command display can be eliminated, and the user can gain additional control over the window appearance.

Functional usage for the title bar should also be extended. File names for layouts and process profiles, for example, can be stored in a menu associated with the title bar, and the user can select the file to be loaded through the mouse control. This method can also provide a convenient way for storing and retrieving data such as process parameters needed for simulations. With the existing display structure, these changes can be made easily.

5.3. Application Interface

One of the main goals with SIMPL-DIX is to be able to predict device characteristics from the process sequence. To accomplish this, connections with additional rigorous device simulators and electrical parameter extractors are required. Interfaces with these external tools can be

established through procedures outlined in this report.

As more simulators become involved, however, a standard process description language should be adopted. In addition, the exchange of data among simulators will be facilitated by the introduction of a profile interchange format (PIF). Thus, in conjunction with the application interface, establishment and utilization of a PIF standard should be a priority for SIMPL-DIX.

5.4. Mask Operations

To explore the interrelationships between layout and process data, more work on internal mask operations is required. In particular, automatic generation of worst case situations for mask bias and misalignment effects are needed. Based on processing parameters and the sequence of fabrication operations, this module should be able to construct HUNCH layers and other layout information needed for process simulations.

In addition, a process critic module which can verify a device topography is needed. Combining with the worst case generator, this design tool should be able to detect topographical problems such as undesired contact between conductors and inadequate thickness of insulator between specified conductors.

References

[1] M. A. Grimm, *SIMPL (SIMulated Profiles from the Layout).* Electronics Research Laboratory, U. C. Berkeley, December 1983.

[2] K. Lee, *SIMPL-2 (SIMulated Profiles from the Layout - version 2).* Electronics Research Laboratory, U. C. Berkeley, July 1985.

[3] W. G. Oldham, A. R. Neureuther, C. Sung, J. L. Reynolds, and S. N. Nandgaonkar, "A General Simulator for VLSI Lithography and Etching Process: Part I - Application to Projection Lithography." *IEEE Trans. Electron. Devices*, Vol. ED-26, No. 4, pp. 717-722, April 1979.

[4] P. Sutardja, Y. Shacham-Diamand, and W. G. Oldham, "Simulation of Stress Effects on Reaction Kinetics and Oxidant Diffusion in Silicon Oxidation." *IEDM 86 Technical Digest*, pp. 526-529, December 1986.

[5] F. R. A. Hopgood, D. A. Duce, E. V. C. Fielding, K. Robinson, and A. S. Williams, *Methodology of Window Management.* Springer-Verlag, 1986.

[6] R. W. Scheifler and J. Gettys, "The X Window System." *ACM Transactions on Graphics*, Vol. 5, No. 2, pp. 79-109, April 1986.

[7] J. Gettys, R. Newman, and T. D. Fera, *Xlib - C Language X Interface (Protocol Version 10).* Massachusetts Institute of Technology, November 1986.

[8] W. M. Newman and R. F. Sproull, *Principles of Interactive Computer Graphics.* McGraw-Hill, 1979.

[9] S. Sechrest, *An Introductory 4.3BSD Interprocess Communication Tutorial.* Computer Science Research Group, Department of Electrical Engineering and Computer Science, U. C. Berkeley, July 1985.

[10] B. W. Kernighan and R. Pike, *The UNIX Programming Environment.* Prentice-Hall, 1984.

[11] S. G. Duvall, "An Interchange Format for Process and Device Simulation." Submitted to *IEEE Trans. on CAD*, January 1987.

[12] B. W. Lindsay and B. T. Preas, "Design Rule Checking and Analysis of IC Mask Designs." *Proc. 13th Design Automation Conference*, pp. 301-308, June 1976.

[13] P. Losleben and K. Thompson, "Topological Analysis for VLSI Circuits." *Proc. 16th Design Automation Conference*, pp. 461-473, June 1979.

[14] E. J. McGrath and T. Whitney, "Design Integrity and Immunity Checking." *Proc. 17th Design Automation Conference*, pp. 263-268, June 1980.

[15] K. Yoshida, "Layout Verification." *Layout Design and Verification*, North-Holland, 1986.

[16] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design.* Addison-Wesley, 1984.

[17] D. S. Harrison, P. Moore, and R. L. Spickelmier, *Data Management and Graphics Editing in the Berkeley Design Environment.* Electronics Research Laboratory, U. C. Berkeley, April 1987.

Appendix A

SIMPL-DIX Manual Page

NAME

SIMPL-DIX – SIMPL Design Interface in X

SYNOPSIS

simpl-dix [options] [CIF files]

DESCRIPTION

SIMPL-DIX is the *X*–window based interactive design interface for running process and device simulators. Currently, it invokes *SIMPL-2* (SIMulated Profiles from the Layout - Version 2) to create the cross-sectional view of a device along an arbitrarily drawn cut-line on the layout. Future release will incorporate additional simulation tools such as *CREEP*.

SIMPL-DIX has a number of internal tools to assist the designer in running process simulation. A layout and profile display editor is provided for the user to define a selected layer, to magnify a selected region, and to obtain doping profile along a selected cut-line. A pattern editor is included which allows the user to add or delete patterns, to modify formats of patterns, and to update pattern specifications. Finally, *HUNCH* is implemented to allow the designer to use operations between masks or sets of masks to highlight locations where topographical problems are anticipated; these problem areas can then be examined using *SIMPL-2* for process verification.

OPTIONS

When *SIMPL-DIX* is invoked, optional command flags can be used to reset default parameters. These options also override those set in the ".Xdefaults" file (see the **X DEFAULTS** section). To restore the default value, a flag can begin with a '**+**' instead of a '**–**'.

The available options are:

–ar Enable the auto-raise mode, which automatically raises the *SIMPL-DIX* window when the mouse cursor enters it.

–b Enable the bell mode, which signals when error occurs.

–bd *color* On color displays, determine the border color of the window. The default is *black*.

–bg *color* On color displays, determine the background color of the window. The default is *white*.

–bw *pixels* Set the width of the window border in pixels. The default is *2 pixels*.

–d Enable the debug mode; all error messages are written to a debug file.

–df *file* Specify the debug file in which the error message is written to, rather than the default */tmp/dixDebug.XXXXX*, where *XXXXX* is the process id of *SIMPL-DIX*. The debug mode is also activated.

–fg *color* On color displays, determine the window foreground (text) color. The default is *black*.

–fm *font* Specify the font to be used for the command menu. The default is *vtbold*. The list of available fonts can be found in the directory "/usr/new/lib/X/font".

–fn *font* Specify the font for the standard text display. The default is *vtsingle*.

–ft *font* Specify the font to be used for the title bar. The default is *vtdwidth*.

–hl *color* On color displays, determine the color to be used for highlighting. The default is *black*.

–if *file* Specify the *SIMPL-DIX* interchange data file, rather than the default */tmp/dixFile.XXXXX*, where *XXXXX* is the *SIMPL-DIX* process id.

–m On color displays, force the display to be in monochrome mode. This is useful in viewing patterns to be printed through the image dumper.

–ms *color* On color displays, determine the color of the mouse cursor. The default is *black*.

–n *name* Specify the name of the *SIMPL-DIX* window to be used by the window manager. This *name* is also displayed in the title bar.

–pf *file* Specify the pattern file to be loaded, rather than the default *SIMPL_pattern*.

–r Reverse definitions of the foreground and background colors.

–rv Same as **–r**.

–tb Enable the title bar with the window name being displayed.

=*geometry* The *SIMPL-DIX* window is created with the specified size and location determined by the supplied *geometry* specification. See *X*(1) for details of this specification.

host:display Normally, *SIMPL-DIX* gets the host and display number to use from the environment variable "DISPLAY". One can, however, specify them explicitly. The *host* specifies which machine to create the *SIMPL-DIX* window on, and the *display* argument specifies the display number. Either value can be defaulted by omission, but ":" is necessary to specify one or both.

X DEFAULTS

SIMPL-DIX allows the user to preset defaults in a customization file called **.Xdefaults** at the user's home directory. The format within the file is "*program_name.keyword:string*", where *program_name* is the local name for *SIMPL-DIX*. See *X*(1) for more details.

Keywords recognized by *SIMPL-DIX* are:

AutoRaise If "*on*", enable the auto-raise mode.

Background Set the background color for color displays.

Bell If "*on*", enable the bell mode.

BodyFont Specify the font for the standard text display.

Border Set the border color for color displays.

BorderWidth Set the border width of the window.

Debug If "*on*", enable the debug mode.

DebugFile Specify the debug file when the debug mode is activated.

Foreground Set the text color for color displays.

Highlight Set the highlight color for color displays.

InterFile Specify the interchange data file to be used with *SIMPL-2*.

MenuFont Specify the font for the command menu.

Mouse Set the mouse cursor color for color displays.

PatternFile Specify the default pattern file.

ReverseVideo Set the reverse-video mode.

SIMPL-2 Specify the *SIMPL-2* path.

TitleBar If "*on*", the title bar is displayed on startup.

TitleFont Specify the font for the title bar.

WindowGeometry Specify the *SIMPL-DIX* window geometry at startup; this enables automatic creation and placement of the window.

WindowName Specify the *SIMPL-DIX* window name.

ENVIRONMENT

DISPLAY To get the default host and display number.

FILES

/cad/bin/simpl-2 Default *SIMPL-2* path.

/usr/new/lib/X/font *X* font directory.

SEE ALSO

X(1), Xlib Documentation
'*SIMPL-2 User Guide*'

AUTHOR

Hsi-Cheng Wu
University of California, Berkeley

Appendix B

Catalog of SIMPL-DIX Routines

SYNOPSIS	*SOURCE FILE*
AddPattern(pattern_id) int pattern_id;	pattern_control.c
AutoAddPattern(pattern_name, pattern_id) char *pattern_name; int pattern_id;	pattern_control.c
CIFAddLayer(new_layer_rt, old_layer_rt) cifLayer *new_layer_rt; cifLayer *old_layer_rt;	cif_utils.c
CIFDoBegin()	cif_action.c
CIFDoBeginCall(symbol_number) int symbol_number;	cif_action.c
CIFDoBeginSymbol(symbol_number, scale_a, scale_b) int symbol_number; int scale_a, scale_b;	cif_action.c
CIFDoBox(length, width, center, direction) int length, width; intPoint center; intPoint direction;	cif_action.c
CIFDoComment(text) char *text;	cif_action.c
CIFDoDeleteSymbol(symbol_number) int symbol_number;	cif_action.c
CIFDoEnd(cif_filename) char *cif_filename;	cif_action.c
CIFDoEndCall()	cif_action.c
CIFDoEndSymbol()	cif_action.c
CIFDoLayer(layer_name) char *layer_name;	cif_action.c
CIFDoPolygon(path) intPath *path;	cif_action.c
CIFDoRoundFlash(diameter, center) int diameter; intPoint center;	cif_action.c
CIFDoTrans(type, point) char type; intPoint point;	cif_action.c

SYNOPSIS	*SOURCE FILE*
CIFDoUserExtension(digit, text) char digit; char *text;	cif_action.c
CIFDoWire(width, path) int width; intPath *path;	cif_action.c
CIFDrawBox(box_rt, viewport, layout_view, pattern_id) cifBox *box_rt; dixViewport viewport; intView layout_view; int pattern_id;	cif_display.c
CIFDrawFlash(flash_rt, viewport, layout_view, pattern_id) cifFlash *flash_rt; dixViewport viewport; intView layout_view; int pattern_id;	cif_display.c
CIFDrawLayer(viewport, layout_view) dixViewport viewport; intView layout_view;	cif_display.c
CIFDrawPolygon(polygon_rt, viewport, layout_view, pattern_id) cifPolygon *polygon_rt; dixViewport viewport; intView layout_view; int pattern_id;	cif_display.c
CIFDrawWire(wire_rt, viewport, layout_view, pattern_id) cifWire *wire_rt; dixViewport viewport; intView layout_view; int pattern_id;	cif_display.c
cifBlock *CIFDupBlock(old_block) cifBlock *old_block;	cif_utils.c
cifBox *CIFDupBox(old_box) cifBox *old_box;	cif_utils.c
cifFlash *CIFDupFlash(old_flash) cifFlash *old_flash;	cif_utils.c
cifLayer *CIFDupLayer(old_layer) cifLayer *old_layer;	cif_utils.c

SYNOPSIS	*SOURCE FILE*
cifPolygon *CIFDupPolygon(old_polygon) cifPolygon *old_polygon;	cif_utils.c
cifWire *CIFDupWire(old_wire) cifWire *old_wire;	cif_utils.c
CIFFlushLayer(layer) cifLayer *layer;	cif_utils.c
CIFFreeBlock(block_ptr) cifBlock *block_ptr;	cif_utils.c
CIFFreeBox(box_ptr) cifBox *box_ptr;	cif_utils.c
CIFFreeFlash(flash_ptr) cifFlash *flash_ptr;	cif_utils.c
CIFFreeLayer(layer_ptr) cifLayer *layer_ptr;	cif_utils.c
CIFFreePolygon(polygon_ptr) cifPolygon *polygon_ptr;	cif_utils.c
CIFFreeSymbol(symbol_ptr) cifSymbol *symbol_ptr;	cif_utils.c
CIFFreeWire(wire_ptr) cifWire *wire_ptr;	cif_utils.c
CIFGetBoxBound(box_ptr, bound_ptr) cifBox *box_ptr; intBound *bound_ptr;	cif_control.c
CIFGetFlashBound(flash_ptr, bound_ptr) cifFlash *flash_ptr; intBound *bound_ptr;	cif_control.c
CIFGetLayoutBound()	cif_control.c
CIFGetPolygonBound(polygon_ptr, bound_ptr) cifPolygon *polygon_ptr; intBound *bound_ptr;	cif_control.c
cifSymbol *CIFGetSymbol(symbol_rt, symbol_number) cifSymbol *symbol_rt; int symbol_number;	cif_utils.c

SYNOPSIS	*SOURCE FILE*
CIFGetWireBound(wire_ptr, bound_ptr) cifWire *wire_ptr; intBound *bound_ptr;	cif_control.c
CIFLoadFile()	cif_control.c
CIFParseBox()	cif_parser.c
CIFParseCallSymbol()	cif_parser.c
CIFParseComment()	cif_parser.c
CIFParseDefineSymbol()	cif_parser.c
CIFParseDeleteSymbol()	cif_parser.c
CIFParseEnd(filename) char *filename;	cif_parser.c
CIFParseFile(filename) char *filename;	cif_parser.c
CIFParseLayer()	cif_parser.c
CIFParsePath(path) intPath **path;	cif_parser.c
CIFParsePoint(point) intPoint *point;	cif_parser.c
CIFParsePolygon()	cif_parser.c
CIFParsePrimitiveCommand()	cif_parser.c
CIFParseRoundFlash()	cif_parser.c
CIFParseUserExtension()	cif_parser.c
CIFParseWire()	cif_parser.c
CIFResizeLayoutBound(bound_box) intBound bound_box;	cif_control.c
CIFScaleBox(box_rt, scale_a, scale_b) cifBox *box_rt; int scale_a, scale_b;	cif_utils.c
CIFScaleFlash(flash_rt, scale_a, scale_b) cifFlash *flash_rt; int scale_a, scale_b;	cif_utils.c

SYNOPSIS	*SOURCE FILE*
CIFScaleLayer(layer_rt, scale_a, scale_b) cifLayer *layer_rt; int scale_a, scale_b;	cif_utils.c
CIFScalePolygon(polygon_rt, scale_a, scale_b) cifPolygon *polygon_rt; int scale_a, scale_b;	cif_utils.c
CIFScaleWire(wire_rt, scale_a, scale_b) cifWire *wire_rt; int scale_a, scale_b;	cif_utils.c
CIFTransBox(box_rt, type, point) cifBox *box_rt; char type; intPoint point;	cif_utils.c
CIFTransFlash(flash_rt, type, point) cifFlash *flash_rt; char type; intPoint point;	cif_utils.c
CIFTransLayer(layer_rt, type, point) cifLayer *layer_rt; char type; intPoint point;	cif_utils.c
CIFTransPolygon(polygon_rt, type, point) cifPolygon *polygon_rt; char type; intPoint point;	cif_utils.c
CIFTransWire(wire_rt, type, point) cifWire *wire_rt; char type; intPoint point;	cif_utils.c
CIFWriteBox(cif_fp, box_rt) FILE *cif_fp; cifBox *box_rt;	cif_control.c
CIFWriteFlash(cif_fp, flash_rt) FILE *cif_fp; cifFlash *flash_rt;	cif_control.c
CIFWriteLayer(cif_fp, layer_ptr) FILE *cif_fp; cifLayer *layer_ptr;	cif_control.c

SYNOPSIS	*SOURCE FILE*
CIFWritePolygon(cif_fp, polygon_rt) FILE *cif_fp; cifPolygon *polygon_rt;	cif_control.c
CIFWriteWire(cif_fp, wire_rt) FILE *cif_fp; cifWire *wire_rt;	cif_control.c
ChangeColor(pattern_id) int pattern_id;	pattern_control.c
ChangeFill(pattern_id, last_x, last_y) int pattern_id; int *last_x, *last_y;	pattern_control.c
ChangePattern(pattern_id) int pattern_id;	pattern_control.c
char *CheckPtr(ptr) char *ptr;	dix_utils.c
ConvertUpper(s) char *s;	dix_utils.c
DefineColor(pattern_id) int pattern_id;	pattern_init.c
DefineFill(pattern_id) int pattern_id;	pattern_init.c
DefineProfileLayer()	profile_control.c
DoDIXAbort()	dix_action1.c
DoDIXEdit()	dix_action1.c
DoDIXReturn()	dix_action1.c
DoHUNCH()	dix_action1.c
DoHUNCHDef()	dix_action1.c
DoHUNCHDefAdd()	dix_action1.c
DoHUNCHDefChange()	dix_action1.c
DoHUNCHDefRemove()	dix_action1.c
DoHUNCHDefSave()	dix_action1.c

SYNOPSIS	*SOURCE FILE*
DoHUNCHDefSet()	dix_action1.c
DoHUNCHList()	dix_action1.c
DoHUNCHLoad()	dix_action1.c
DoHUNCHSave()	dix_action1.c
DoLayout()	dix_action1.c
DoLayoutCut()	dix_action1.c
DoLayoutFrame()	dix_action1.c
DoLayoutLoad()	dix_action1.c
DoLayoutZoom()	dix_action1.c
DoPattern()	dix_action2.c
DoPatternAdd()	dix_action2.c
DoPatternChange()	dix_action2.c
DoPatternMove()	dix_action2.c
DoPatternRemove()	dix_action2.c
DoPatternSave()	dix_action2.c
DoProfile()	dix_action2.c
DoProfileDoping()	dix_action2.c
DoProfileFrame()	dix_action2.c
DoProfileLayer()	dix_action2.c
DoProfileLoad()	dix_action2.c
DoProfileZoom()	dix_action2.c
DoSIMPL()	dix_action2.c
DoSIMPLBatch()	dix_action2.c
DoSIMPLInput()	dix_action2.c
DoSIMPLInputDepo()	dix_action2.c
DoSIMPLInputDevl()	dix_action2.c

SYNOPSIS	*SOURCE FILE*
DoSIMPLInputEtch()	dix_action2.c
DoSIMPLInputExpo()	dix_action2.c
DoSIMPLInputImpl()	dix_action2.c
DoSIMPLInputOxid()	dix_action2.c
DoSIMPLOpenClosePC()	dix_action2.c
DoSIMPLQuit()	dix_action2.c
DoSIMPLSave()	dix_action2.c
DrawColorFill(viewport, pattern_id) int pattern_id;	pattern_display.c
DrawColorMap(viewport, pattern_id) dixViewport viewport; int pattern_id;	pattern_display.c
DrawCommandBox(command_id, foreground, background) int command_id; Pixel foreground, background;	command_control.c
DrawCommandMenu()	command_control.c
DrawDopingProfile()	profile_control.c
DrawDopingScale()	profile_control.c
DrawFillMap(viewport, pattern_id) dixViewport viewport; int pattern_id;	pattern_display.c
DrawFilledBox(viewport, x, y, width, height, foreground, background) dixViewport viewport; int x, y, width, height; Pixmap foreground, background;	graphics_control.c
DrawFilledPolygon(viewport, vertex_list, vertex_size, foreground, background) dixViewport viewport; Vertex *vertex_list; int vertex_size; Pixmap foreground, background;	graphics_control.c
DrawLayoutCutline(viewport, layout_view) dixViewport viewport; intView layout_view;	cutline_control.c

SYNOPSIS	*SOURCE FILE*
DrawLayoutFrame(viewport, layout_view) dixViewport viewport; intView layout_view;	frame_control.c
DrawLeftBar()	scrollbar_control.c
DrawLeftPatterns()	pattern_display.c
DrawOutlinedBox(viewport, x, y, width, height, color, func) dixViewport viewport; int x, y, width, height; Pixel color; int func;	graphics_control.c
DrawOutlinedPolygon(viewport, vertex_list, vertex_size, color, func) dixViewport viewport; Vertex *vertex_list; int vertex_size; Pixel color; int func;	graphics_control.c
DrawPattern(pattern_id) int pattern_id;	pattern_display.c
DrawProfileCutline(viewport) dixViewport viewport;	cutline_control.c
DrawProfileFrame(viewport, profile_view) dixViewport viewport; floatView profile_view;	frame_control.c
DrawRightBar()	scrollbar_control.c
DrawRightPatterns()	pattern_display.c
DrawRubberbandBox(viewport, x1, y1, x2, y2, color) dixViewport viewport; int *x1, *y1, *x2, *y2; Pixel color;	graphics_control.c
DrawRubberbandLine(viewport, x1, y1, x2, y2, color) dixViewport viewport; int *x1, *y1, *x2, *y2; Pixel color;	graphics_control.c
DrawSolidBox(viewport, x, y, width, height, color, func) dixViewport viewport; int x, y, width, height; Pixel color; int func;	graphics_control.c

SYNOPSIS	*SOURCE FILE*
DrawSolidPolygon(viewport, vertex_list, vertex_size, color, func) dixViewport viewport; Vertex *vertex_list; int vertex_size; Pixel color; int func;	graphics_control.c
DrawTextBar()	scrollbar_control.c
floatPath *DupFloatPath(old_path) floatPath *old_path;	dix_utils.c
intPath *DupIntPath(old_path) intPath *old_path;	dix_utils.c
ExitError(error_message, error_code) char *error_message; int error_code;	dix_utils.c
hunchGeo *FANGToHUNCH(fang_geo) fa_geometry *fang_geo;	hunch_operation.c
float FitFloatPoint(x, x1, x2, y1, y2) float x; float x1, x2; float y1, y2;	dix_utils.c
float FitJunctionDoping(delta, dist, doping) float delta; float dist, doping;	profile_control.c
FlushLayoutCutline()	cutline_control.c
FlushLayoutView(layout_view) intView *layout_view;	view_control.c
FlushProfileCutline()	cutline_control.c
FlushProfileView(profile_view) floatView *profile_view;	view_control.c
FreeDopingSpectrum()	doping_control.c
FreeFloatPath(path_ptr) floatPath *path_ptr;	dix_utils.c

SYNOPSIS	*SOURCE FILE*
FreeIntPath(path_ptr) intPath *path_ptr;	dix_utils.c
GetAnswer(message, field_size) char *message; int field_size;	prompt_control.c
GetCommandId()	command_control.c
GetDefaults(dix_name) char *dix_name;	dix_main.c
GetEvent(request) unsigned long request;	display_control.c
GetFileName(message, file_control) char *message; int file_control;	prompt_control.c
flaotPoint *GetFloatIntersect(p1, p2, q1, q2) floatPoint p1, p2, q1, q2;	dix_utils.c
GetLayoutCutline()	cutline_control.c
GetLayoutFrame()	frame_control.c
GetLayoutViewX(viewport, layout_view, layout_x) dixViewport viewport; intView layout_view; int layout_x;	view_control.c
GetLayoutViewY(viewport, layout_view, layout_y) dixViewport viewport; intView layout_view; int layout_y;	view_control.c
GetLayoutX(viewport, layout_view, viewport_x) dixViewport viewport; intView layout_view; int viewport_x;	view_control.c
GetLayoutY(viewport, layout_view, viewport_y) dixViewport viewport; intView layout_view; int viewport_y;	view_control.c
GetPatternId(pattern_name) char *pattern_name;	pattern_init.c
GetProfileCutline()	cutline_control.c

SYNOPSIS	*SOURCE FILE*
float GetProfileDoping(selected_pt) floatPoint selected_pt;	profile_control.c
GetProfileFrame()	frame_control.c
simplPolygon *GetProfilePolygon(selected_pt) floatPoint selected_pt;	profile_control.c
GetProfileViewX(viewport, profile_view, profile_x) dixViewport viewport; floatView profile_view; float profile_x;	view_control.c
GetProfileViewY(viewport, profile_view, profile_y) dixViewport viewport; floatView profile_view; float profile_y;	view_control.c
float GetProfileX(viewport, profile_view, viewport_x) dixViewport viewport; floatView profile_view; int viewport_x;	view_control.c
float GetProfileY(viewport, profile_view, viewport_y) dixViewport viewport; floatView profile_view; int viewport_y;	view_control.c
dixSpectrum *GetSpectrumPtr(spectrum_ptr, level) dixSpectrum *spectrum_ptr; float level;	doping_control.c
GetTextWidth(text, size, font_info) char *text; int size; FontInfo *font_info;	graphics_control.c
GetYesOrNo(message) char *message;	prompt_control.c
HUNCHAddBox(geo_ptr, left, bottom, right, top) hunchGeo *geo_ptr; int left, bottom; int right, top;	hunch_utils.c

SYNOPSIS	*SOURCE FILE*
hunchLayer *HUNCHAddLayerDef()	hunch_control.c
HUNCHAddLayerGeo(layer_ptr) hunchLayer *layer_ptr;	hunch_control.c
HUNCHChangeLayerDef(old_layer) hunchLayer *old_layer;	hunch_control.c
HUNCHClearLayerGeo()	hunch_control.c
HUNCHDoArg(name) char *name;	hunch_action.c
HUNCHDoBegin(filename) char *filename;	hunch_action.c
HUNCHDoBeginDef(name) char *name;	hunch_action.c
HUNCHDoBinaryOp(op_code) int op_code;	hunch_action.c
HUNCHDoClear()	hunch_action.c
HUNCHDoClose()	hunch_action.c
HUNCHDoComment(text) char *text;	hunch_action.c
HUNCHDoEnd()	hunch_action.c
HUNCHDoEndDef()	hunch_action.c
HUNCHDoError(error_code) int error_code;	hunch_action.c
HUNCHDoMoveBack()	hunch_action.c
HUNCHDoOp(op_code) int op_code;	hunch_action.c
HUNCHDoOpen()	hunch_action.c
HUNCHDoSeparator()	hunch_action.c
HUNCHDoUnaryOp(op_code) int op_code;	hunch_action.c

SYNOPSIS	*SOURCE FILE*
HUNCHDrawGeo(geo_rt, viewport, layout_view, pattern_id) hunchGeo *geo_rt; dixViewport viewport; intView layout_view; int pattern_id;	hunch_display.c
HUNCHDrawLayer(viewport, layout_view) dixViewport viewport; intView layout_view;	hunch_display.c
hunchBox *HUNCHDupBox(old_box) hunchBox *old_box;	hunch_utils.c
hunchGeo *HUNCHDupGeo(old_geo) hunchGeo *old_geo;	hunch_utils.c
HUNCHFlushArg(arg_ptr) hunchArg *arg_ptr;	hunch_utils.c
HUNCHFlushGeo(geo_ptr) hunchGeo *geo_ptr;	hunch_utils.c
HUNCHFlushLayer(layer_ptr) hunchLayer *layer_ptr;	hunch_utils.c
HUNCHFlushOp(op_ptr) hunchOp *op_ptr;	hunch_utils.c
HUNCHFreeArg(arg_ptr) hunchArg *arg_ptr;	hunch_utils.c
HUNCHFreeBox(box_ptr) hunchBox *box_ptr;	hunch_utils.c
HUNCHFreeComment(comment_ptr) hunchComment *comment_ptr;	hunch_utils.c
HUNCHFreeGeo(geo_ptr) hunchGeo *geo_ptr;	hunch_utils.c
HUNCHFreeLayer(layer_ptr) hunchLayer *layer_ptr;	hunch_utils.c
HUNCHFreeOp(op_ptr) hunchOp *op_ptr;	hunch_utils.c

SYNOPSIS	*SOURCE FILE*
hunchGeo *HUNCHGetCIFGeo(cif_layer) cifLayer *cif_layer;	hunch_layer.c
HUNCHGetDef(op_ptr, def_string) hunchOp *op_ptr; char *def_string;	hunch_utils.c
hunchGeo *HUNCHGetGeo(layer_name) char *layer_name;	hunch_layer.c
HUNCHListDef()	hunch_display.c
HUNCHLoadFile()	hunch_control.c
hunchGeo *HUNCHMakeGeo(op_ptr) hunchOp *op_ptr;	hunch_layer.c
HUNCHMakeLayerGeo()	hunch_control.c
hunchGeo *HUNCHOpAnd(hunch_geo1, hunch_geo2) hunchGeo *hunch_geo1; hunchGeo *hunch_geo2;	hunch_operation.c
hunchGeo *HUNCHOpBloat(hunch_geo, amount) hunchGeo *hunch_geo; int amount;	hunch_operation.c
hunchGeo *HUNCHOpBound(hunch_geo) hunchGeo *hunch_geo;	hunch_operation.c
hunchGeo *HUNCHOpIn(geo1, geo2) hunchGeo *geo1; hunchGeo *geo2;	hunch_operation.c
hunchGeo *HUNCHOpNand(geo1, geo2) hunchGeo *geo1; hunchGeo *geo2;	hunch_operation.c
hunchGeo *HUNCHOpNor(geo1, geo2) hunchGeo *geo1; hunchGeo *geo2;	hunch_operation.c

SYNOPSIS	*SOURCE FILE*
hunchGeo *HUNCHOpNot(hunch_geo) hunchGeo *hunch_geo;	hunch_operation.c
hunchGeo *HUNCHOpOr(hunch_geo1, hunch_geo2) hunchGeo *hunch_geo1; hunchGeo *hunch_geo2;	hunch_operation.c
hunchGeo *HUNCHOpOut(geo1, geo2) hunchGeo *geo1; hunchGeo *geo2;	hunch_operation.c
hunchGeo *HUNCHOpScale(hunch_geo, scale) hunchGeo *hunch_geo; float scale;	hunch_operation.c
hunchGeo *HUNCHOpSub(hunch_geo1, hunch_geo2) hunchGeo *hunch_geo1; hunchGeo *hunch_geo2;	hunch_operation.c
hunchGeo *HUNCHOpTouch(geo1, geo2) hunchGeo *geo1; hunchGeo *geo2;	hunch_operation.c
hunchGeo *HUNCHOpXor(hunch_geo1, hunch_geo2) hunchGeo *hunch_geo1; hunchGeo *hunch_geo2;	hunch_operation.c
hunchGeo *HUNCHOpXTrans(geo, amount) hunchGeo *geo; int amount;	hunch_operation.c
hunchGeo *HUNCHOpYTrans(geo, amount) hunchGeo *geo; int amount;	hunch_operation.c
HUNCHParseArg()	hunch_parser.c
HUNCHParseComment()	hunch_parser.c
HUNCHParseDef()	hunch_parser.c

SYNOPSIS	*SOURCE FILE*
HUNCHParseFile(filename) char *filename;	hunch_parser.c
HUNCHParseOp()	hunch_parser.c
HUNCHRemoveLayerDef(del_layer) hunchLayer *del_layer;	hunch_control.c
HUNCHSaveLayerDef()	hunch_control.c
HUNCHSaveLayerGeo()	hunch_control.c
hunchLayer *HUNCHSelectDef(message) char *message;	hunch_display.c
fa_geometry *HUNCHToFANG(hunch_geo) hunchGeo *hunch_geo;	hunch_operation.c
HighlightPattern(pattern_id, foreground, background) int pattern_id; Pixel foreground, background;	pattern_display.c
HighlightViewport(viewport) dixViewport viewport;	display_control.c
InitCursors()	display_init.c
InitDisplay(dix_name) char *dix_name;	display_init.c
InitFonts()	display_init.c
InitMenu()	command_control.c
InitPatternBars()	scrollbar_control.c
InitPatterns()	pattern_init.c
InitTextBar()	scrollbar_control.c
MakeDopingSpectrum()	doping_control.c
MovePattern(src_id, dst_id) int src_id, dst_id;	pattern_control.c
PFEscapeWhiteSpace(white_space_control, file_control) int white_space_control; int file_control;	parser_control.c

SYNOPSIS	*SOURCE FILE*
PFGetCharacter(white_space_control, file_control) int white_space_control; int file_control;	parser_control.c
PFGetInteger(white_space_control, file_control) int white_space_control; int file_control;	parser_control.c
PFGetString(white_space_control, file_control) int white_space_control; int file_control;	parser_control.c
PFLookAheadFor(c) char c;	parser_control.c
PSEscapeWhiteSpace(white_space_control, string_control) int white_space_control; int string_control;	parser_control.c
PSGetCharacter(white_space_control, string_control) int white_space_control; int string_control;	parser_control.c
PSGetToken(white_space_control, string_control) int white_space_control; int string_control;	parser_control.c
PSLookAheadFor(c) char c;	parser_control.c
PrintCenteredText(viewport, x, y, width, height, text, font_info, foreground, background, text_func) dixViewport viewport; int x, y, width, height; char *text; FontInfo *font_info; Pixel foreground, background; int text_func;	graphics_control.c
PrintLayoutCutline()	cutline_control.c
PrintMessage(viewport, message, font_info) dixViewport viewport; char *message; FontInfo *font_info;	prompt_control.c
PrintProfileCutline()	cutline_control.c

SYNOPSIS	*SOURCE FILE*
PrintText(viewport, x, y, width, height, text, font_info, foreground, background, text_func) dixViewport viewport; int x, y, width, height; char *text; FontInfo *font_info; Pixel foreground, background; int text_func;	graphics_control.c
PrintUsage(dix_name) char *dix_name;	dix_utils.c
Prompt(message) char *message;	prompt_control.c
Redraw()	display_control.c
RemovePattern(pattern_id) int pattern_id;	pattern_control.c
RingBell(volume) int volume;	display_control.c
SIMPLClosePC()	simpl_interface.c
SIMPLConnect()	simpl_interface.c
SIMPLCopyDoping(old_doping, new_doping) float old_doping[GRID_X_SIZE][GRID_Y_SIZE]; float new_doping[GRID_X_SIZE][GRID_Y_SIZE];	simpl_utils.c
SIMPLCopyGridX(old_grid_x, new_grid_x) float old_grid_x[GRID_X_SIZE]; float new_grid_x[GRID_X_SIZE];	simpl_utils.c
SIMPLCopyGridY(old_grid_y, new_grid_y) float old_grid_y[GRID_Y_SIZE]; float new_grid_y[GRID_Y_SIZE];	simpl_utils.c
SIMPLCountGridX(profile_grid_x) float profile_grid_x[GRID_X_SIZE];	simpl_utils.c
SIMPLCountGridY(profile_grid_y) float profile_grid_y[GRID_Y_SIZE];	simpl_utils.c
SIMPLCountPolygon(polygon_rt) simplPolygon *polygon_rt;	simpl_utils.c
SIMPLCountVertex(polygon_ptr) simplPolygon *polygon_ptr;	simpl_utils.c

SYNOPSIS	*SOURCE FILE*
SIMPLDrawAir(viewport, profile_view) dixViewport viewport; floatView profile_view;	simpl_display.c
SIMPLDrawGrid(viewport, profile_view) dixViewport viewport; floatView profile_view;	simpl_display.c
SIMPLDrawPolygon(viewport, profile_view) dixViewport viewport; floatView profile_view;	simpl_display.c
SIMPLDrawProfile(viewport, profile_view) dixViewport viewport; floatView profile_view;	simpl_display.c
SIMPLErrorHandler()	simpl_interface.c
SIMPLFlushGrid(profile_grid_x, profile_grid_y, profile_doping) float profile_grid_x[GRID_X_SIZE]; float profile_grid_y[GRID_Y_SIZE]; float profile_doping[GRID_X_SIZE][GRID_Y_SIZE];	simpl_utils.c
SIMPLFreePolygon(polygon_ptr) simplPolygon *polygon_ptr;	simpl_utils.c
SIMPLGetAnswer(message) char *message;	simpl_interface.c
SIMPLGetRequest()	simpl_interface.c
SIMPLLoadFile()	simpl_control.c
SIMPLNewProfile()	simpl_interface.c
SIMPLOldProfile()	simpl_interface.c
SIMPLOpenPC()	simpl_interface.c
SIMPLQuit()	simpl_interface.c
SIMPLRead()	simpl_interface.c
SIMPLReadGrid(profile_fp) FILE *profile_fp;	simpl_control.c
SIMPLReadLayout(profile_fp) FILE *profile_fp;	simpl_control.c
SIMPLReadPolygon(profile_fp) FILE *profile_fp;	simpl_control.c

SYNOPSIS	*SOURCE FILE*
SIMPLReadProfile(profile_name) char *profile_name;	simpl_control.c
SIMPLReset()	simpl_interface.c
SIMPLRunBatch()	simpl_interface.c
SIMPLRunInput(process) char *process;	simpl_interface.c
SIMPLSaveProfile()	simpl_control.c
SIMPLSet()	simpl_interface.c
SIMPLWrite()	simpl_interface.c
SIMPLWritePCHeader()	simpl_interface.c
SIMPLWriteProfile(filename) char *filename;	simpl_control.c
SavePattern()	pattern_control.c
ScaleProfileDist(viewport, dist) dixViewport viewport; float dist;	profile_control.c
ScaleProfileDoping(viewport, doping) dixViewport viewport; float doping;	profile_control.c
SelectCommand()	command_control.c
SelectPattern(message, edit_mode, move_mode) char *message; short edit_mode, move_mode;	pattern_control.c
SetCommand(menu_id, command_id, name_0, name_1, name_2, proc) int menu_id, command_id; char *name_0, *name_1, *name_2; int (*proc)();	command_control.c
SetMenu(menu_id, menu_size) int menu_id, menu_size;	command_control.c
SetPatterns()	pattern_init.c
SetViewports()	display_init.c
UpdateBarSize()	scrollbar_control.c

SYNOPSIS	*SOURCE FILE*
UpdateLayoutDisplay(viewport, layout_view) dixViewport viewport; intView layout_view;	display_control.c
UpdateLeftBar(y) int y;	scrollbar_control.c
UpdateProfileDisplay(viewport, profile_view) dixViewport viewport; floatView profile_view;	display_control.c
UpdateRightBar(y) int y;	scrollbar_control.c
UpdateTextBar(selected_x, selected_y) int selected_x, selected_y;	scrollbar_control.c
WriteError(error_message) char *error_message;	prompt_control.c
WriteTitle()	display_control.c
WriteXError(display, error_ptr) Display *display; XErrorEvent *error_ptr;	display_init.c
ZoomLayout()	view_control.c
ZoomProfile()	view_control.c
main(argc, argv) int argc; char *argv[];	dix_main.c