A View on Components of Software and Application Programming in Distributed Environments

THESIS

Presented to the Graduate Council of

Southwest Texas State University

in Partial Fulfillment of the Requirements

for the degree of

MASTER OF SCIENCE

by

Jeffrey Wijono

SOUTHWEST TEXAS STATE UNIVERSITY

August, 1992

Acknowledgments

I would like to thank a number of people who have helped me in the process of writing this thesis. First, my thesis advisor, Dr. Carol Hazlewood, who has helped me a tremendous amount in finishing this thesis paper. Her comments changed the shape of this paper and improved it significantly. Special thanks are due to Mr. Zoran Nevajdic, who made this research happen. He gave me the idea for this topic and helped me solve various programming problems when they arose. He also gave me much encouragement in keeping up with the programming. Third, Professor Donald Hazlewood, who has generously let me use the computers facility in the Southwest Texas State University mathematics laboratory, which made the research possible. Fourth, to my committee members, Dr. Thomas McCabe and Dr. Grady Early who have spent time in reviewing and giving suggestions to the paper. Lastly, I would like to thank Ms. Debra Goldberg, who has helped me in reviewing the English grammar throughout this paper.

Jeffrey Wijono

Abstract

A View on Components of Software and Application Programming in Distributed Environments

We describe a view of application design in the modern programming environment in which reusability of software, network transparent display, and processing, vertically and horizontally, play an important role. We provide a toolkit for creating applications. We address reusability and probe the ways for adding to or customizing the functionality of existing applications.

The toolkit consists of a transformation library, a communication library, and a control and display library. The transformation library converts the output of applications to a standard form. The user is free to manipulate the data which is then normally sent to a display process for display.

For example, we can consider a UNIX command like 'ps -ef' as an application. The output of this application displays the user ID, process ID, parent process ID, Stime, TTY, time, and command to the standard output. The control library is used to invoke the command repeatedly, with a given frequency, for the purpose of monitoring the processes. By using the transformation library, the output of this application will be converted to a standard form, a table that consists of rows and columns. A row represents a process and a column represents one of the attributes of the ps -ef command. Using this table the user program will enhance this UNIX command working in concert with the display process which has been created using functions from the display library which currently, as an example, consists of a few process independent *widgets* created using Motif toolkit. All communication is accomplished using functions from the communication library.

Table Of Contents

Acknowledgment i

Abstract ii

Table of Contents iii

List of Figures iv

Chapter 1 Introduction 1

Chapter 2 User Interface 7

Chapter 3 Reusable Software 9

Chapter 4 Programming Environment 12

Chapter 5 Description of Libraries 18

Chapter 6 Purpose of Libraries 49

Chapter 7 Applications 53

Chapter 8 Conclusions and Further Research 66

Bibliography 68

Apendices A Applications' code A1

Apendices B Communication Library's Code B1

Apendices C Control Library's Code C1

Apendices D Transform Library's Code D1

Apendices E Display Library's Code E1

Apendices F Miscellaneous Library's Code F1

List of Figures

Fig. 1-1 Application with different processes 2
Fig. 1-2 Distributed Application 2
Fig. 1-3 Distributed Application with Fault Tolerance 3
Fig. 1-4 Complex Application 4
Fig. 4.1 Xt-Based Toolkits 16
Fig. 6-1 Libraries Relations 49
Fig. 7-0 Application - Libraries Relation 53
Fig. 7-1 Main Window 54
Fig. 7-2 File Save Pop-Up Window 56
Fig. 7-3 Command Pop-Up Window 58
Fig. 7-4 Delay Pop-Up Window 59
Fig. 7-5 Key Pop Up Window 61
Fig. 7-6 Column Pop-Up Window 62

Chapter 1

Introduction

The primary purpose of this research is to explore a view of application design with attention to some of the major building blocks surfacing in the modern computing environment. The emphasis is on process independence of components, objects and encapsulation of code, reusability, creation of user interfaces, and communication transparency with respect to processing and data presentation.

Encapsulation of code (object oriented programming) hides some of the complexities of the code from the user. Encapsulation can provide a reusable, simplified interface to a complex system like X-Windows, fault tolerance, distributed system, etc. Thus, encapsulation reduces the software development time.

Reusability of software is a very important aspect in software development. There are many software companies that are creating reusable software nowadays. Some of these companies are exclusively making reusable modules that can be used by other companies.

In the past few years we have witnessed the rapid development of the user interface. It has been recognized that a powerful and expressive user interface improves the efficiency of the user, whether he happens to be a developer or end user.

Exchange of information and communication between processes marks another aspect of computation that is growing rapidly. The demand and performance expectations for fault tolerant distributed applications and network transparencies are increasing.

This research identifies problems in some specific modern application that has the above characteristic. The folowing pictures are samples of modern applications nowadays. These pictures are only at design level now, which help this study to identify some possible problems that arise for programming such applications.

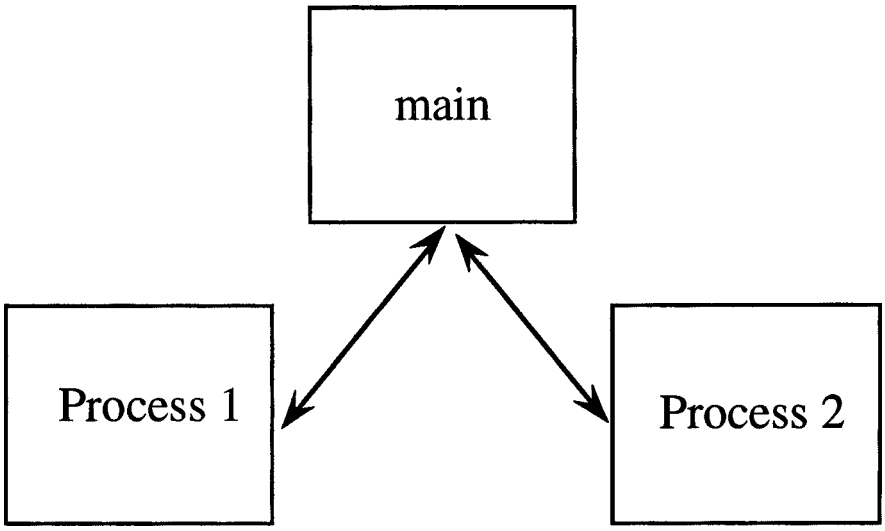

Fig. 1-1 Application with different Processes

Fig. 1-1 shows an example of an application with different processes on the same machine. The main process needs to communicate with process1 and process2 through the communication channel.

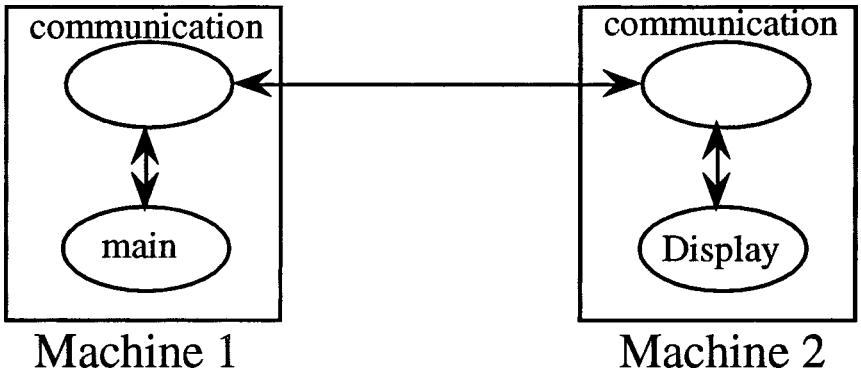

Fig. 1-2 Distributed Application

Fig. 1-2 shows an example of an application with tasks that run on different machines to achieve a better performance.

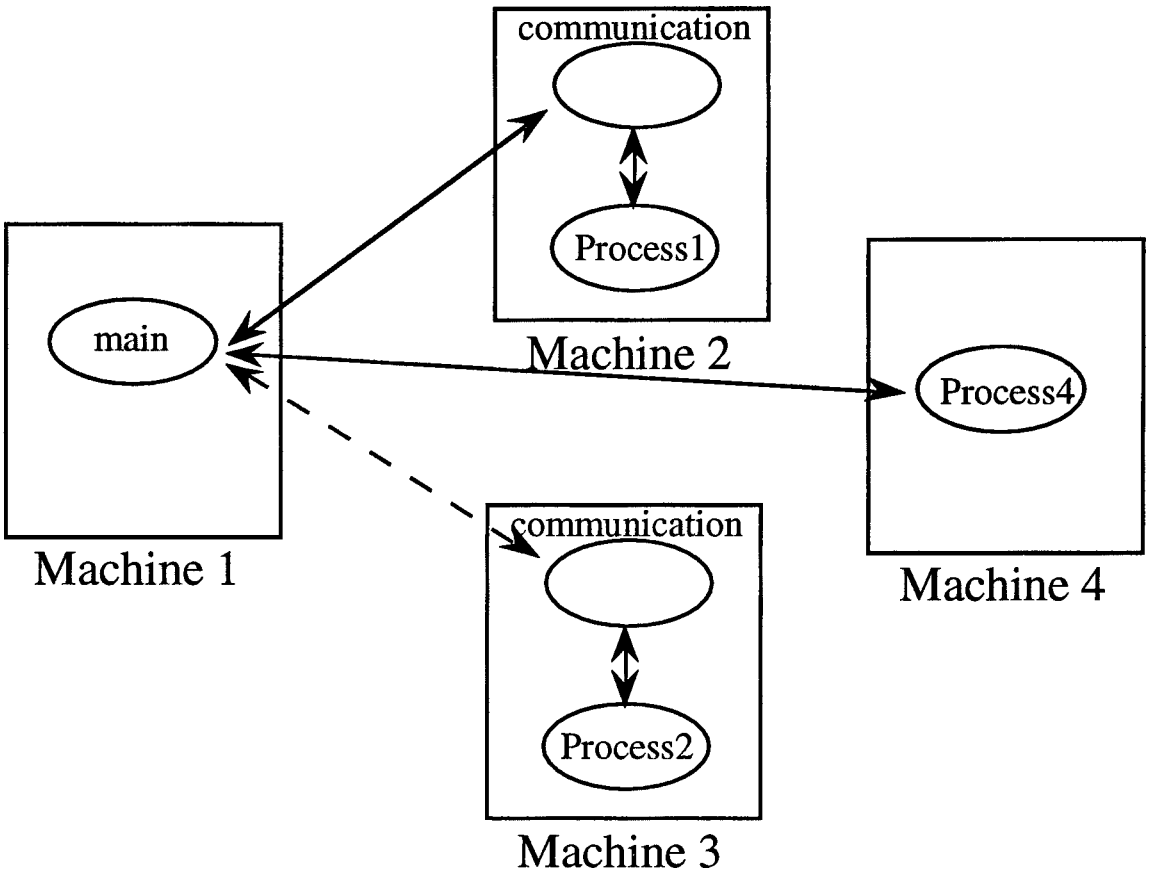

Fig. 1-3 Distributed Application with Fault Tolerance

Fig. 1-3 shows an example of an application with tasks on different machines. The communication between machine1 and machine3 is only established if machine2 is unavailable or fails. A fault-tolerance system will automatically move the job at machine2 to machine3 if machine2 fails so the application can still run normally.

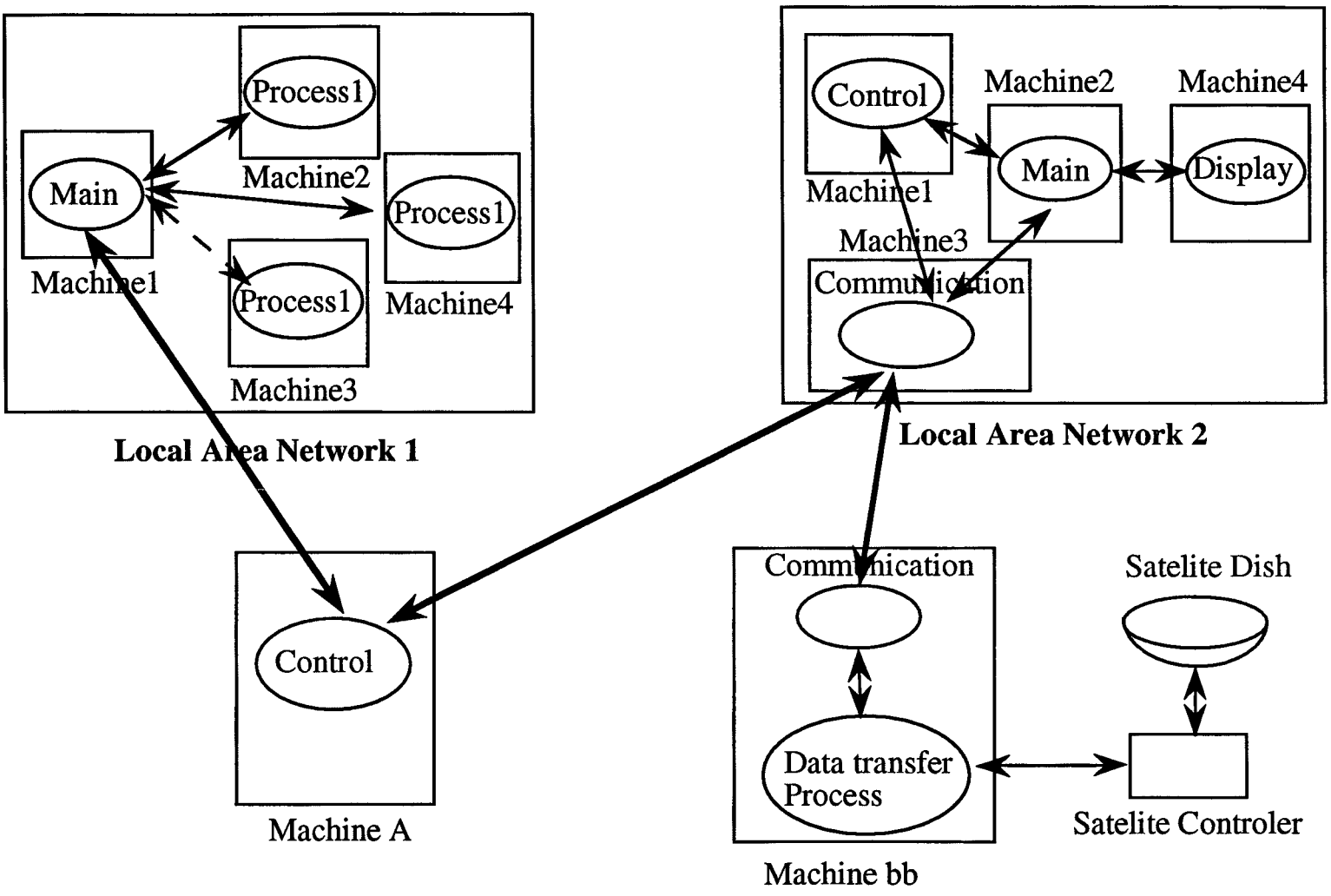

Fig. 1-4 Complex Application

During this exploration, we wrote some supporting code that we have divided into five categories. First, the communication library provides communication functions for information exchange and message passing between processes in the UNIX system, such as messages passing between parent and child process. Second, the transformation library consists of the transformation of a table form into a row-column form. Third, the control library adds the functionalities of the row-column form. Fourth, the display library allows the creation of an output display using the MOTIF User Interface on X-Window. The last library consists of some miscellaneous functions that are useful for general purposes; such as functions for creating a linked list or deleting a linked list. These functions are used in the four previous libraries, but they do not really belong to any of those libraries.

1.1 Objective

The objective of this research is to produce a new toolkit that can help programmers develop new applications in a short period of time and develop a toolkit that improves the effectiveness of a reusable component library in creating a specific type of application mentioned in the introduction. The approach followed was to research the techniques and experimental applications associated with reusable software libraries, communication processes, and user interface such as Motif Toolkit, Xt Toolkit, and Xlib itself.

1.2 Organization

This paper is divided into eight chapters. The first chapter contains an introduction to this research and the organization of the paper. Chapter 2 presents the basic concept of a user interface and a brief history of user interfaces. Chapter 3 covers the basic concept of reusability, the advantages of writing a reusable module, and the programming aspect of writing a reusable module. Chapter 4 introduces the programming environment in this research. It starts with an introduction to X Window, its history, its basic concept, and the

Motif User Interface. Chapter 5 contains a brief description of each library and functions in the library. Chapter 6 describes the programming technique for a specific example of the these new libraries. Chapter 7 contains the User Manual for applications created in this research. These applications are examples of how one can use the libraries. Chapter 8 concludes this study and provides some suggestions for further research.

Chapter 2

User Interface

A user interface (UI) is an interface between a human and a computer; a method used by either an operating system or an application program to interact with the user. The user interface is the way in which a user can communicate with the computer. An interface includes the computer itself with its keyboard, screen, mouse, commands, menus, programming language, and the way a program is presented to the screen [19].

A user interface (UI) is responsible for accepting all input from a user and for displaying the output from the computer to the user. Today's user interface uses state of the art methods such as windows, pull down menus, pop up menus, dialog boxes, and on-line help [4]. One important component of a user interface is interaction between a user and the Central Processing Unit (CPU). When the user performs any action on the computer, such as issuing a command, selecting an object, or moving the mouse, the UI will provide some sort of response to the screen. For example, when a user issues a command, the user interface will supply a response to the user; this response can be different for each command.

The first interface between a computer and a human was punched cards. The computer read the input from the cards and printed the result to the cards again in batch mode. There was not any interaction between the computer and the users.

The original interactive user interface was the command-line interface (CLI). Users communicated with computer systems through teletypewriters (TTYs), character-based terminals that could accept only typed input and could print only on paper, one character after another. The most familiar CLI today is the 'DOS A>' prompt.

As technology improved, video display terminals (VDTs) became widely available. These "glass TTYs" could position a character anywhere on the screen.

Nowadays there is a need for a good user interface. There are several companies that have started selling programming toolboxes with ready-made functions that can be used to create a variety of applications [4].

According to Lamb, there has been considerable interest in the user interfaces of software systems in the last decade, because a user interface is the portion of the system that presents information to and accepts input from the user, and because a user interface defines the observable behavior of a system [10].

The main function of a user interface is communication. The interface must keep the computer from coming between the user and the work and it must be consistent throughout the program. Also, the user interface must be flexible to satisfy a wide range of user requirements. A good user interface will provide on-line help.

Chapter 3

Reusable Software

According to Hooper and Chester, there are two possible definitions of software reusability: first, the extent to which a software component can be used (with or without adaptation) in multiple problems solutions; second, the extent to which a software component can be used (with or without adaptation) in a problem solution other than the one for which it was originally developed [6].

Reusable software is a software module that can be used in more than one computer program. According to Hooper and Chester [6], "Reuse is the process by which existing software work products (which may include not only source code, but also products such as documentation, design, test data, tools, and specifications) are carried over and used in a new development effort, preferably with minimal modification."

Reuse does not just encompass the reuse of code; it is possible to reuse specifications and designs. It has been suggested that code reuse is never likely to be cost effective. The cost of developing reusable software could be more than developing regular software since it might require modification to the software component. However it will be cheaper in the long run because of reusability. Moreover there are other important benefits of writing reusable modules such as:

- Acceleration of software production

  Reusable modules can be used in other programs than it was originally designed for.
- Reduction in the cost of software development

  Reusing software saves time in designing and debugging the software.
- Improvement in the software quality

Reusing a module in another program will reduce the possibility of having bugs in the program since the reusable module has been used and tested already.

- Reduction of software development time:

  It is often the case that bringing a system to market as early as possible is more important than overall development costs. Reusing a software module speeds up system production because both development and testing time are reduced.

- Increase in system reliability:

  Only actual operational use adequately tests components and reused components.

- Reduction of overall risks:

  There is less uncertainty in the costs of using the existing components than in the costs of developing them. This is an important factor for project management as it reduces the overall uncertainty in the project cost estimation.

One of the important aspects of reusable software is portability. Portability is a characteristic of software closely related to reusability. A portable software component is one that can be used in multiple machine environments, where the physical hardware, operating system, run-time environment, and compiler conventions may all vary. Thus, portability is necessary to achieve reusability across multiple machine environments [6].

The term module has a variety of meanings in computer science. Sometimes module is used as a synonym for procedure or function. In this research, module refers to a collection of functions and data that are manipulated as a unit. We divide a program into collections of related functions and data structures; each of these collections is stored in a file and can be compiled separately (compiled module).

This research will analyze the modularity of an application with the possibility of reusing the module; the research will generate a toolkit that can be used for creating applications. In this manner, existing code can be reused, and perhaps more importantly,

changes in application code can be accommodated without modifying (and recompiling) the existing library.

Chapter 4

Programming Environment

X Window

X Window is a window system that has a capability of being network transparent. The terms of client and server in the world of X means different things than the terms of client and server in the Apple or regular PC world. In the Apple networks, a server is generally a shared device that delivers some kind of services, such as access to files, applications, or printers, to the network's clients or users. While in the X Window world, the clients are hardware-independent applications that let you access either locally (on the same machine) or remotely (on another machine the network using TCP/IP protocol). The X server contains programs specific to particular workstations or terminals that can control and draw the display when they receive requests or commands from clients. The X server's role is fairly passive; it listens for commands from clients or input from users (such as key presses and mouse clicks) [1].

The X Window system requires a window manager, an X application which intervenes between the client and the server to tell the server how to put information from the client into a scrollable, resizeable window. It also includes the desktop manager, which tells the window manager how to put such a window onto the icon base. Most of the window managers and desktop managers in X are generally separate from the X window itself. For example, there are X.desktop from IXI, and HP-vue from Hewlett-Packard/SAIC for the desktop manager; and for the window manager, there are commercial software such as, Motif from the Open Software Foundation, OPEN LOOK from AT&T and Sun Microsystems; or even public domain window manager such as Tom's Window Manager [1].

X-Window is a window system that can accommodate multiple applications simultaneously, generating text and graphics in monochrome or color on a bitmapped display. X-Window is network transparent; that is, it can display applications running on different machines as if they are running on one machine.

The X-window system is descended from an earlier window system called W. The W-window was designed with a synchronous protocol. In the summer of 1984, Robert Scheifler of the MIT Laboratory for Computer Science replaced the synchronous protocol with an asynchronous protocol, and replaced the display lists with immediate mode graphics.

The design and implementation of the first ten versions of X were primarily the work of three individuals: Robert Scheifler and James Gettys of Digital Equipment Corporation, and Ron Newman of MIT. The design was done mainly under the MIT/Project Athena. Robert Scheifler started to develop X-Window in 1984 while he was working at MIT's Laboratory for Computer Science (LCS) on a distributed system called Argus. There was a need for a decent display environment for debugging multiple distributed processes at that time. Jim Gettys, a Digital Equipment Corporation (DEC) Engineer, was assigned to MIT's Project Athena, an undergraduate education program sponsored by Digital and IBM that would populate the MIT campus with thousands of workstations.

However, X version 11 is the result of the effort of many individuals at many locations and organizations. The first release of version 11 (X11 R1) was made available on September 15, 1987, release 2 in March 1988, release 3 in February 1989, and release 4 in January 1990.

X allows a program to display windows containing text and graphics on any hardware that supports the X-protocol without modifying, recompiling, or relinking the application. An application need not be running on the same machine that actually supports

the display. One of the most important features of X is its unique device-independent architecture. This device independence allows X-based applications to function in a heterogeneous environment consisting of mainframes, workstations, and personal computers [20].

Another unique feature is that X does not define any particular user interface style. X provides mechanisms to support many interface styles rather than enforcing one policy. Therefore, the basic X Window System does not provide user interface components such as menus, dialog boxes, and button boxes. Most X-applications depend on higher level libraries built on top of the X-library. Such libraries provide a familiar procedural interface that masks the details of the protocol encoding and transport interactions, and automatically handles the buffering of requests for efficient transport to the server. There are X libraries provided for different languages; for example, Xlib is the library for the C programming language [12].

X supports asynchronous operation; clients use requests to register interests in various events while the server sends the event notification asynchronously. In this case clients do not have to wait for a response from the server before generating a new request. The communication protocol is designed to be independent of the operating system in order to support distributed computing in a heterogeneous environment.

Disadvantages of X are that X applications require much memory and speed in order to work properly, and it is very costly to the system because every command is executed as a network round-trip.

The X server creates and manipulates windows in response to requests from clients. When the client and the server reside on the same machine, communication between them is often implemented using shared memory. If they reside on different machines, communication between them can take place over any network transparent layer that provides reliability in transferring data in both directions [12].

MOTIF User Interface

OSF/Motif is a graphical user interface combining a toolkit, presentation description language, window manager, and style guide. A user can save time in learning a new Motif application, since much of the knowledge gained from using other Motif applications translates directly to the new application. The Motif User Interface is not a piece of software, it is rather a specification for a user interface that is built on top of the X-window graphical user interface. According to Johnson and Reichard, authors of three X-Window Programming books [7], [8], [9], there are three reasons for using Motif:

1. Motif provides a standard interface with a consistent look and feel. Users will have less work to do in learning how to interact with a new Motif Applications.
2. Motif provides a very high level object-oriented library. The whole idea of using Motif is to reuse the functions in the toolkit. Therefore, it can reduce or save valuable time in writing code.
3. Motif has been adopted by many of the major players in the computer industry.

Motif provides a toolkit that is easier to use than the Xt toolkit. Using functions in the toolkit can save valuable time in creating Motif applications.

The Xt intrinsic provides some basic mechanisms for many widget sets. The Xt intrinsic is built on top of the X Window system. The Motif Toolkit is built on top of the Xt toolkit because it uses many of the Xt features and functions. However, Motif has a set of functions of its own, and it has its own widget set with its own look and feel.

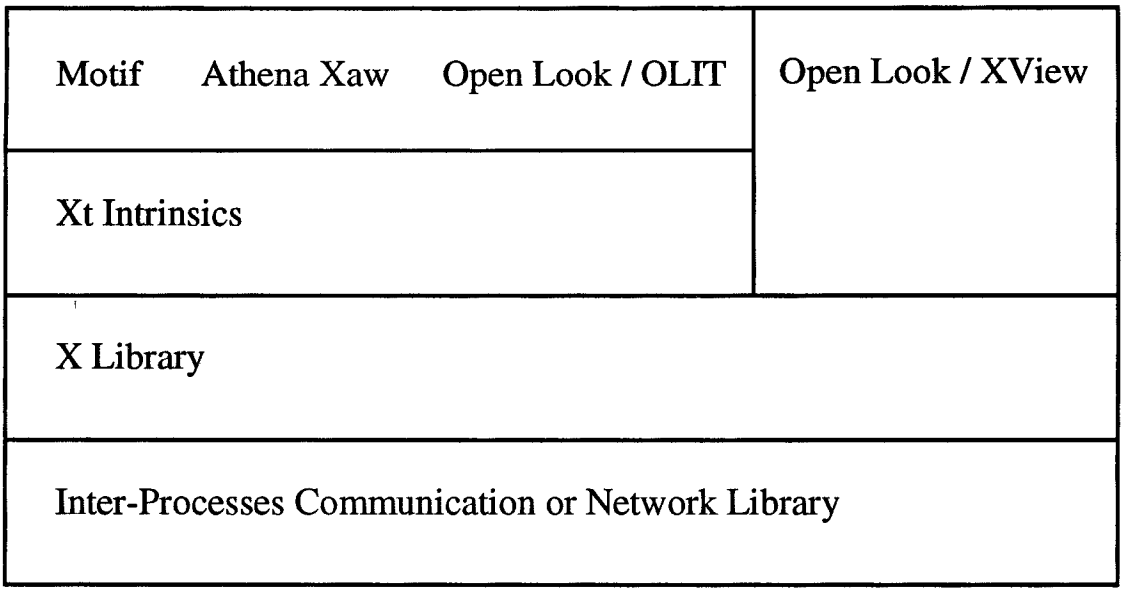

Fig. 4-1 Xt-Based Toolkits

A widget is a complex data structure that combines an X window with a set of procedures that perform actions on that window (X window along with some procedures that manipulate the window). A widget structure contains an ID of the X window used by the widget and additional data needed by these procedures. Widgets in Motif are designed so that many of their resources can be modified during runtime. For instance, a user can change the background color, height, width, or position of the widget by changing the value in the app-defaults file [6]; the user may change the value of these widgets without recompiling the program. App-defaults file is a file that Xt automatically loads when the application is being run. Usually the file is stored in the directory /usr/lib/X11/app-defaults and an app-defaults file name is the same as the application's name with the first letter being a capital.

There are several widgets sets that are freely available as contributed software: Hewlett Packard's X widget set (Xw), Sony's Xsw widget set, the Athena widget sets (Xaw) developed by MIT/Project Athena, and the Cornell widget set. There are also some

proprietary widget sets available. The Open Software Foundation (OSF) and its member companies distribute the Motif widget set, and AT&T markets the XT+ toolkit which includes a widget set, based on the OPEN LOOK interface style.

Chapter 5

Description of Libraries

5.1 Communication Library

This library contains functions that manage the communication between processes in the UNIX environment. This library supports the message passing in the UNIX environment using pipe, shared memory, and messages. The message is passed as a linked list. First, the sender function will send the number (N) of nodes in the list to the communication channel; then the sender function loops N times to send each node in the list. For each node transfer, the sender initially sends the length of the string that will be sent through the communication channel, and then it sends the string to the communication channel. The receiver function always tries to read the number (N) of nodes that it will receive. If N is greater than zero, then the receiver function loops until it reads the N nodes from the communication channel. Otherwise, the receiver function will skip the loop.

5.1.1 Data Structures and Global Variables in the Communication Library

Data Structures:

```
typedef struct _shared_mem_pipe_struct {
        char  *addr;          /* shared memory block address */
        int    semid;         /* semaphore key */
        int    segid;         /* segment ID  */
        } shared_mem_pipe;
```

Global Variables:

none

5.1.2 Description of Functions in the Communication Library

read_shared_mem(fds, buf, nbytes)

int fds, nbytes;

char *buf;

This function is similar to the functionality of the standard read function on the UNIX system except this function will read 'nbytes' characters from the shared memory. The function starts reading 'nbytes' characters from the shared memory if the semaphore is free and it returns the number of characters read from the shared memory buffer. Otherwise, the function returns a zero number if the semaphore is not free.

write_shared_mem(fds, buf, nbytes)

int fds, nbytes;

char *buf;

This function is similar to the functionality of the standard write function on the UNIX system except this function will write 'nbytes' characters from the shared memory. The function starts writing 'nbytes' characters from the shared memory if the semaphore is free and returns the number of characters successfully written into the shared memory buffer. Otherwise, this function return a zero number if the semaphore is held by other processes.

create_shared_mem_pipe(fds)

int *fds;

This function initializes the shared memory communication channel in the UNIX system. It sets up all the neccesary buffers and semaphore to be used by read_shared_mem and write_shared_mem.

close_shared_mem_pipe(fds)

int *fds;

This function releases the memory used by the shared memory communication channel. This function must be called before exiting from the program. Unlike the regular file descriptor, the shared memory communication does not close by itself when the program exits.

read_message(fds, buf, size)

int fds, size;

char *buf;

This function is similar to the functionality of the standard read function on the UNIX system except this function will read nbytes characters from 'fds' using the mesage communication in the UNIX system. The function starts reading 'size' characters from the message (communication channel) if it is available and returns the number of characters read from the message. Otherwise, the function returns a zero number if the message is empty.

write_message(fds, buf, size)

int fds, size;

char *buf;

This function is similar to the functionality of the standard write function on the UNIX system except this function will write nbytes characters from 'fds' using message communication in the UNIX system. The function starts writing nbytes characters from the message if it is available and returns the number of characters written into the message buffer. Otherwise, the function will return zero if the message is full.

create_message_pipe(fd)

int *fd;

This function initializes the message communication channel in the UNIX system. It sets up all the neccesary buffers to be used by read_message and write_message.

close_message_pipe(fd)

int *fd;

This function releases the buffer used by the message communication channel. This function must be called before exiting from the program. Unlike the regular file descriptor, the message communication does not close by itself when the program exits.

create_pipe(fd)

int *fd;

This function releases the communication channel using the regular UNIX pipe communication system.

close_pipe(fd)

This function closes the file descriptor that is used for the communication channel with the pipe communication system.

set_file_descriptor_no_delay(fd)

int *fd;

This function sets the file desciptor on both ends of the pipe so there is no delay. This will allows the process to do other things when the file descriptor is busy.

safe_read(fds, buf, nbytes)

int fds, nbytes;

char *buf;

This function reads nbytes characters from the communication channel 'fds', or it will not read anything. The function returns zero or nbytes. It returns zero if it does not read anything. It returns nbytes if it reads something, because the function will keep reading from the communication channel 'fds' until it reads nbytes characters. The function will use the standard read function in the UNIX system for the pipe communication. For the shared memory communication it will use the read_shared_mem function, and the message system will use read_message function.

safe_write(fds, buf, nbytes)

int fds, nbytes;

char *buf;

This function will return zero if the file descriptor is full, but it will wait an indefinite amount of time until it can successfully write nbytes characters if it started to write some bytes of character into the communication channel 'fds'. The function will use

the standard write function in the UNIX system for the pipe communication. For the shared memory communication it will use the write_shared_mem function, and the message system will use write_message function.

read_list(fds, head)

int fds;

node_struct **head;

This function reads the stream of data from the communication channel 'fds', and creates the linked list of data. This function returns the head of the linked list and it guarantees to read all the data in the linked list; therefore, it cannot be interupted by other processes once it starts reading a character from the communication channel 'fds'.

write_list(fds, head)

int fds;

node_struct **head;

This function writes the stream of data in the linked list of data into the communication channel 'fds'. This function guarantees to write all the data into the list; therefore, it cannot be interrupted by other processes once it starts writing a character into the communication channel 'fds'.

read_str (fds, str, ch)

int fds;

char str[], ch;

This function reads a stream of characters into str from the communication channel designated by 'fds'. Reading terminates when the character in variable ch is

encountered. This function will not wait for a character in the fds file descriptor. If the 'fds' is empty, then 'str' will return NULL.

read_list_delay(fds, head, delay)

int fds, delay;

node_struct **head;

This function loops for a specified number of times ('delay'), calling the read_list function, and pausing for one second. If the read_list function returns a number greater than zero (something is read from the communication channel) then it will break the loop.

5.2 Control Library

This library contains functions for executing a UNIX command and creating a table form output. The output of this UNIX command will be sent to the main program through the communication channel. The communication can be either pipe, shared memory, or messages that are already available on UNIX system V. All the functions in this library can access a global variable called control_dat.

Global variable in this library:
control_dat_struct *control_dat;

5.2.1 Data Structures and Global Variables in the Control Libraries

Data structures:

```
typedef struct control_data {
        char command[30];     /* Command to be executed */
        char key[40];         /* keyword matching */
        int heading;          /* boolean if the command will have heading line*/
        int delay;            /* duration time of the command to be executed */
        int col_n;            /* column of key to be matched */
        int col[MAX_COLUMN+1];        /* selected column to be displayed */
        int fd_cntl_in[2];    /* file descriptor number for communication to */
        int fd_cntl_out[2];   /* file descriptor number for communication from */
        int (*read_function)(); /* pointer to the read function */
        int (*write_function)(); /* pointer to the write function */
        } control_data_struct;
```

Global Variables:

control_data_struct *control_dat;

5.2.2 Description of Functions in the Control Library

monitor ()

This function is responsible for preparing the communication channel (pipe) between this process and the main program. First the function will fork the main process. The child process invokes the 'process' function to perform the UNIX command and sends the output to the main program through the pipe (communication channel). It will do this repeatedly until it receives a terminate command from the main program. The parent process returns to the main program and continues performing some other tasks.

monitor_sm ()

This function is identical to the above monitor function, except this function will use the shared memory system instead of the pipe system for the communication channel.

monitor_message ()

This function is identical to the above monitor function, except this function will use the message system instead of the pipe system for the communication channel.

process(head, pause)

node_struct *head;

int *pause;

This function accepts some commands from the main program through the communication channel; for instance, changing the UNIX command or the delay time, sending a pause or resume signal to the process. If the process is not in the pause mode then it calls the monitor_system function to perform the UNIX command.

monitor_system(command)

char *command;

This function performs the UNIX command and sends the output of this command to the main program. First the function creates a pipe, then forks the process. The child process will close the standard output and duplicate the standard output of this process to the writing pipe, and then this process performs the UNIX command. The main process of this function reads the output of the child process from the reading pipe and sends the result of this output to the main program through the communication channel.

receive_from_control(head)

node_struct **head;

This function reads the linked list of data from the communication channel between the monitor process and the main program. This function returns the head of the linked list which contains the information that has just been read from the communication channel. The purpose of this function is to hide all the complexity of the control data structure and communication channels between the control process and the main process.

send_to_control(head)

node_struct **head;

This function sends the linked list of data to the communication channel between the monitor process and the main program. This function receives the head of the linked list which contains the information that needs to be written to the communication channel as a function parameter. The purpose of this function is to hide all the complexity of the control data structure and communication channels between the control process and the main process..

5.3 Transform Library

The transform library contains functions that will transform a table form into row-column form. The column is generated by the physical location of each column. First the function will scan throughout the table and split the table into rows. For every row the function will compute the beginning and ending location of each column using a distance formula. The distance formula is used to figure out where the current column belongs by comparing the beginning and ending of the current column to the previous and to the next column.

5.3.1 Data Structures and Global Variables in the Transform Library

Data Structures:

```
typedef struct loc {
        char name[50];          /* title of the column */
        int min, max;           /* min and max position of each column */
        } loc_struct;

typedef struct column_data {
        char *content;                  /* the content of each column */
        struct column_data *next_col;   /* pointer to the next column */
        } column_data_struct;

typedef struct row_col {
        column_data_struct *col_head;   /* column pointer */
        struct row_col *next_row;       /* pointer to the next row */
```

```
} row_col_struct;

typedef struct trans_data {
        int    fd_trans[2];                    /* file desriptor */
        int    col[MAX_COLUMN+1];              /* column to be displayed */
        void   (*func)();    /* function to be invoked when the menu is selected */
        row_col_struct *row_col_head;          /* pointer to the head of data */
        } trans_data_struct;

typedef struct _pid_list {
        int_node          *list;          /* pointer to the list of pid list struct */
        struct _pid_list  *next;          /* pointer to the next node on the list */
        } pid_list_struct;
```

Global Variables:

none

5.3.2 Description of Functions in the Transform Library

row_col_struct ***trans_to_row_col**(head)
node_struct **head;

This function transforms the table form into the row-column form. First the function calls the get_first_loc() with the node in the 'head', to figure out the number of columns in the table and the location of each column. The get_max_loc() functions figures out the final location of each column in the table. Once it figures

out the location of each column, then it parses the table into the row-column form and returns the head of this row-column form to the caller of this function.

node_struct ***create_display_list**(row_col_head, key, cols, col_of_key)
row_col_struct *row_col_head;
char *key;
int cols[MAX_COLUMN+1], col_of_key;

This function creates a final list that will be sent to the display list.

void **get_first_loc** (str, loc, num)
char str[];
struct loc loc;
int num;

This function computes the beginning and ending position of each column on the first row of the table and stores the location into a variable called 'loc'. It also computes the number of columns from the title heading of the table into the integer variable called 'num'.

void **get_max_loc** (str, loc, num)
char str[];
struct loc loc;
int num;

This function computes the beginning and ending position of each column in the table and stores it into a variable called 'loc' by going through the whole table using the distance formula. The beginning and ending of each column is stored in the variable 'loc'.

int **distance** (start, last, loc, i)

int start, last, i;

struct loc;

The function computes the distance between two strings. The distance will be negative if one of the strings overlaps the other.

void **parse** (key, col_n, num2, n_of_col, head, location, fdp)

char key[];

int col_n, num2[MAX_COLUMN], n_of_col, fdp;

struct loc loc;

This function parses the table output into row-column form, using variable 'key', 'col_n', and 'num2' to select some columns in the table. It compares the variable 'key' with the 'n_of_col'; if they match, then this function sends the selected column into the communication channel.

match_pid(head, pid)

int_node *head;

int pid;

This function matches the variable 'pid' in the linked list that is pointed to by 'head'. This function returns True if it finds the matching 'pid', or it returns False if it does not find the matching 'pid' in the list.

str_to_array(str, cols)

char *str;

int cols[MAX_COLUMN+1];

This function converts a string of characters into an array of integers with one or more white spaces as a delimiter between integers on the string.

void skipblanks (str1, str2)

char *str1, *str2;

Copy 'str1' into 'str2' removing all leading white space.

5.4 Display Library

The display library contains functions for users to create output windows and menus in the X-Window with the Motif User Interface. The library is divided into two parts: the first part contains functions for creating the output display, and for managing the communication between these windows and processes. The second part of this library contains functions for creating the menu of the window and for managing all functions that need to be invoked when the menu is selected. The display windows can be either text widget, list widget, or graph widget.

5.4.1 Data Structures and Global Gariables in the Display Library

Data Structures:

```
Typedef struct _menu_data {
        char    *name;              /* name on menubar */
        void    (*func)();          /* callback function */
        int     n_sub_items;        /* number of items at the submenu */
        char    *sub_menu_title;    /* title of sub_menu */
        struct _menu_data *next;    /* pointer to the next menu in the menubar */
        struct _menu_data *sub_menu; /* pointer to the submenu item */
        } menu_struct;

typedef struct _menu_list {
        menu_struct     *menu;      /* name of the current menu */
        struct _menu_list  *next;   /* pointer to the next menu list on the list */
        } menu_list_struct;
```

```
typedef struct _display_data {
        Widget  toplevel;               /* toplevel Widget */
        Widget  aForm;                  /* after the toplevel */
        Widget  menubar;                /* menubar widget */
        int     fd_disp_in[2];          /* file descriptor from main */
        int     fd_disp_out[2];         /* file descriptor out to main */
        int     (*read_function)();     /* pointer to the reading function */
        int     (*write_function)();    /* pointer to the writing function */
        int     (*close_communication)(); /* pointer to close communication */
        XtWorkProcId work_ proc_id;
        } display_win_struct;

typedef struct _display_list {
        Widget  title;                  /* title display */
        Widget  display;                /* scrolled list */
        Widget  vscroll;                /* vertical scrolled bar */
        Widget  hscroll;                /* horizontal scrolled bar */
        int     pause;                  /* pause condition */
        } display_list_struct;

typedef struct _display_text {
        Widget  title;                  /* title display */
        Widget  display;                /* scrolled list */
        Widget  vscroll;                /* vertical scrolled bar */
        Widget  hscroll;                /* horizontal scrolled bar */
```

```
        int         pause;              /* pause condition */
        } display_text_struct;

typedef struct _graph_data {
        Widget      title;              /* title display */
        Widget      canvas;             /* graphics display window */
        char        title_name[100];    /* title name of the current display */
        int         depth;              /* number of depth screen */
        int         ncolors;            /* number of color */
        int         count;              /* number of data */
        int         granulation;        /* number of bar on the graph windows */
        int         scale               /* the maximum number for the height of bar */
        int         foreground;         /* foreground color */
        int         background;         /* background color */
        int_node *head;                 /* pointer to the data list */
        GC          gc;                 /* graphic content */
        Dimension width;                /* width of the graph window */
        Dimension height;               /* height of the graph window */
        Pixmap pix;                     /* buffer for the graph display */
        } display_graph_struct;

typedef struct _dialog_data{
        Widget      label;              /* parent widget of this dialog */
        Widget      value[3];           /* widget for the value of this dialog */
        int         num_of_col;         /* the current number of columns */
        short int   type;               /* type of display widget (text/list) */
```

```
} dialog_data_struct;
```

Global Variables:

```
display_list_struct     *display_list_dat;
display_text_struct     *display_text_dat;
display_graph_struct    *display_graph_dat;
```

5.4.2 Description of Functions in the Display Library

menu(str, func)

```
char  *str;
void  (*func)();
```

This function creates a pull-down menu by creating a submenu list as the child of the current menu. The variable 'str' is the title of the pull-down menu, and the variable 'func' is a pointer to the function to be invoked when the menu is selected. Every call to menu must be matched by a call to done_menu.

done_menu()

This function adds a NULL node at the end of the submenu list. This function must be called to end the menu list (NULL terminates the submenu list that is created by the menu function.) Every call to menu must be matched by a call to done_menu.

menu_item(str, func)

```
char  *str;
void  (* func)();
```

This function assigns items in the pull-down menu. The variable 'str' is the title of the menu item, and the variable 'func' is a pointer to the function to be invoked when the item is selected.

create_menu_buttons(title, menu, menulist)

char *title;

Widget menu;

menu_struct *menulist;

This function creates pull-down menus from the information in the global variable 'menudat '. The function uses XmCreatePulldownMenu to create a pull-down menu and registers the appropriate callback function for every item on the menu.

CreateMenubar(display_list_dat, name)

display_list_data_struct *display_list_dat;

char *name;

This function returns a label widget for the menubar with all items registered on the menubar.

CreateTextMenubar(display_list_dat, name)

display_list_data_struct *display_list_dat;

char *name;

This function returns a text widget for the menubar with all items registered on the menubar.

void DoSaveList(w, m_data, call_data)

Widget w;

menu_data_struct *m_data;

XmAnyCallbackStruct call_data;

This function saves the information on the List Widget into a file. It first creates the pop-up dialog window and prompts for a file name.

void DoSaveText(w, m_data, call_data)

Widget w;

menu_data_struct *m_data;

XmAnyCallbackStruct call_data;

This function saves the information on the Text Widget into a file. It first creates the pop-up dialog window and prompts for a file name.

void DoPrintList(w, m_data, call_data)

Widget w;

menu_data_struct *m_data;

XmAnyCallbackStruct call_data;

This function gets the information of the List widget and sends it to the standard printer output.

void DoPrintText(w, m_data, call_data)

Widget w;

menu_data_struct *m_data;

XmAnyCallbackStruct call_data;

This function gets the information of the Text widget and sends it to the standard printer output.

void DoQuit(w, file_quit, call_data)

Widget w;

menu_data_struct *file_quit;

XmAnyCallbackStruct call_data;

This function sends the "_TERMINATE" message to the main program.

DoCommand(parent, co_comand)

Widget parent;

menu_data_struct *co_command;

This function creates a pop-up window for the user to enter a new UNIX command. This function allows the application user to change the UNIX command at any time. The new UNIX command is sent to the main program through the communication channel.

DoDelay();

This function creates a pop-up dialog window and prompts a new duration time. It allows the user to set the duration time of the UNIX command that will be executed. The new value of delay time is sent to the main program through the communication channel.

DoKeySelect(parent, disp_key)

Widget parent;

menu_data_struct *disp_key;

This function creates a pop-up dialog window and prompts a keyword and a column number for matching the keyword on any specified column. The desired keyword

and the column number for this keyword is sent to the main program through the communication channel.

DoRowSelect(parent, disp_key)

Widget parent;

menu_data_struct *disp_key;

This function generates a list of the row numbers that are selected on the list widget. This list is sent to the main program through the communication channel.

DoColSelect(parent, disp_key)

Widget parent;

menu_data_struct *disp_key;

This function creates a pop-up dialog window and displays the toggle button widget of the column heading. A list of the column number that is selected is sent to the main program through the communication channel.

DoSnap()

This function snaps the current display to another display process.

DoGranulation()

This function sets the number of bars on the graph display.

DoScale()

This function sets the number of maximum height of bar in the graph display.

void **initial_list_display**()

This function assigns the default value for the global variable display_list_dat.

void **initial_text_display**()

This function assigns the default value for the global variable display_text_dat.

void **create_display_pipe()**

This function assigns the communication function to the pipe system.

void **create_display_shared_memory_pipe()**

This function assigns the communication function to the shared memory system.

void **create_display_message_pipe()**

This function assigns the communication function to the message system.

void **create_list_display**(argc, argv)

int argc;

char *argv[];

This function creates a display window. It creates a list widget for the main output window and calls CreateMenubar to create the menu for this window.

void **create_text_display**(argc, argv)

int argc;

char *argv[];

This function creates a display window. It creates a text widget for the main output window and calls CreateMenubar to create the menu for this window.

void **create_graph_display**(argc, argv)

int argc;

char *argv[];

This function creates a display window. It creates a graph widget for the main output window and calls CreateMenubar function to create the menu for this window.

void display_title(w, fd_title, id)

Widget w;

int *fd_title;

XtInputId *id;

This function reads a string from the file descriptor file_buffer and displays it to the title widget.

void **display_list_data**(dsp, fd_display, id)

Widget dsp;

int *fd_display;

XtInputId *id;

This function uses the read_list function in the communication library to read the information from the main program and sends the result into the list display window.

void **display_text_data**(dsp, fd_display, id)

Widget dsp;

int *fd_display;

XtInputId *id;

This function uses the read_list function in the communication library to read the information from the main program and sends the result into the text display window.

void **display_graph_data**(dsp, fd_display, id)
Widget dsp;
int *fd_display;
XtInputId *id;

This function uses the read_list function in the communication library to read the information from the main program and sends the result into the graph display window.

receive_from_display(head)
node_struct *head;

This function reads the linked list of data from the communication channel between the display process and the main program. This function returns the head of the linked list which contains the information that has just been read from the communication channel. The purpose of this function is to hide all the complexity of the display data structure and communication channels between the display process and the main process.

send_command_to_display(head)
node_struct **head;

This function sends the linked list of commands to the communication channel between the display process and the main program. This function receives the head of the linked list which contains the information that needs to be written to the

communication channel as a function parameter. The purpose of this function is to hide all the complexity of the display data structure and communication channels between the control process and the main program.

send_data_to_display(head)

node_struct **head;

This function sends the linked list of data to the communication channel between the display process and the main program. This function receives the head of the linked list which contains the information that needs to be written to the communication channel as a function parameter. The purpose of this function is to hide all the complexity of the display data structure communication channels between the control process and the main process.

send_block_to_display(buf, size)

This function sends a block of data to the communication channel between the display process and the main program.

5.5 Miscellaneous Library

This library contains some generic functions that will be used by other functions in other libraries. The reason for separating these functions from the communication, control, transform, and display libraries is that they are used repeatedly in these libraries (such as function for creating/inserting and erasing a linked list.) They are used in the communication library, transform library, and control library.

5.5.1 Data Structures and Global Variables in the Miscellaneous Library

Data Structures:

```
typedef struct node {
        char        *line;      /* data line */
        struct node *next;      /* pointer to next node struct */
        } node_struct;

typedef struct _int {
        int         num;        /* integer node */
        struct _int *next;      /* pointer to next int_ node struct */
        } int_node;
```

Global Variables:

none

5.5.2 Description of Functions in the Miscellaneous Library

insert_node(head, new)

node_struct *head, *new;

This function inserts a new node into the list. Head is a pointer to the beginning of the list. If the head is NULL (the list is empty), then it inserts the new node at the beginning of the list, or else it inserts the new node at the end of the list. A new node is a row in the table of output.

clear_node(head)

node_struct **head;

This function releases all the memory that is used by the node_struct linked list pointed by the variable 'head'.

insert_int_str(head, str)

int_node **head;

char *str;

This function converts the string variable 'str' into integer then inserts it at the end of the integer list.

insert_int_node(head, num)

indt_node **head;

int num;

This function inserts the integer variable 'num' into the end of the integer list.

clear_int_node(head)

int_node **head;

This function releases all the memory that is used by the int_node linked list pointed to by variable head.

char ***e_malloc**(size)

int size;

This function performs the regular malloc, and it will exit the program if it fails to malloc memory.

syserr(msg)

char *msg;

This function prints msg to the standard error file and computes the error number.

Chapter 6

Purpose of Libraries

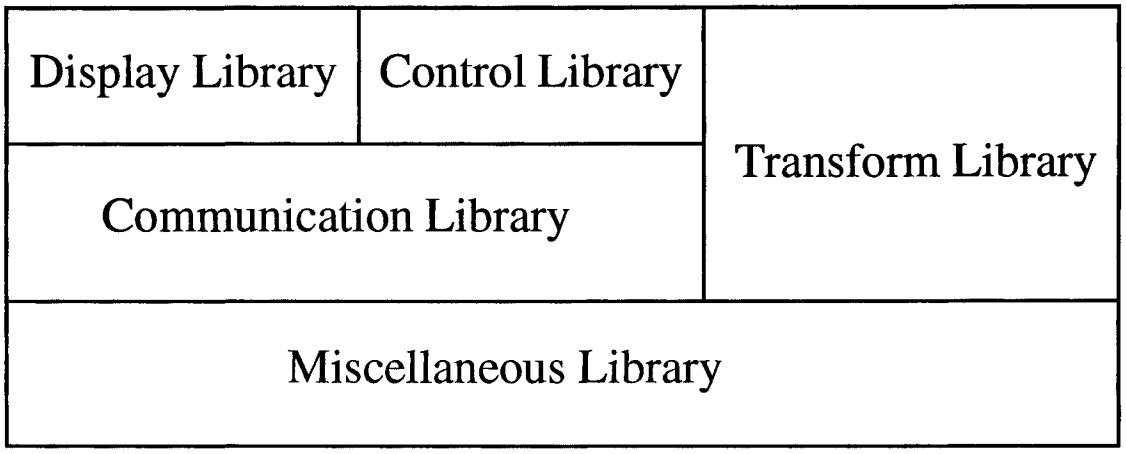

Fig. 6-1 Libraries Relation

6.1 Communication Library

Communication is an important component in future computing development since the computer is becoming more advanced than ever before. Networking is widely available today. However, there is no standard library of functions with communication in mind. Programmers always have to spend time studying a different type of communication for almost every type of communication or distributed system of networking. The difference can sometimes lead to frustration since one system can be very different from another.

We try to minimize the different structures and complexity of different types of communication by creating a communication library that consists of uniform parameters for the functions and a simple data structure. This library provides functions for message passing and data transfer between processes, such as communication using pipe, shared

memory and messages. These functions are designed to help programmers transfer messages between processes.

The purpose of this library is to provide uniform functions for all type of message passing (communication) between processes. For instance, the library provides functions for communication using pipes, messages, and shared memory by simulating read(fds, buf, nbytes) and write(fds, buf, nbytes), which are standard C functions in the UNIX communication using pipes. These functions are read_message(fds, buf, nbytes), write_message(fds, buf, nbytes), read_shared_mem(fds, buf, nbytes), and write_shared_mem(fds, buf, nbytes). They all have the same parameters as the original read and write functions.

This library should be extended with some additional functions that will allow communication between machines. These are some of the functions that can be added to the library: communication though serial lines, communication between machines on the network using ethernet with the TCP/IP protocol, or even some newer protocols that can allow communication between different systems, such as Appletalk.

One can imagine encapsulating these communiation functions into **communication objects** which could run on a variety of machines being able to communicate between each other under a number of protocols.

6.2 Control Library

The control library was intended to provide functions which can be used to manage and control processes. As an example, we implement a monitor function which monitors (executes) a UNIX command with a given delay and aquires its output. This data can then be manipulated by the user.

6.3 Transformation Library

There is always a need to compute or generate different types of input and output of programs. For instance, one might desire to transform the output from one program into another form so it can be used as input to another program. Usually a packet received though a network channel follows a certain format, which might not be a format that one wants. Sometimes we need to transform the data into a certain form so we can transfer it into the communication channel. For instance, the read_list and write_list from the communication library will only transfer a linked list in which each node is a NULL terminated string (string definition in C language).

The transformation library should contain functions which help the user to transform data from one form into another. As an example, we have row-column transform. First, the function breaks the table form into rows, then it uses the physical location of the column in the table to break the rows into columns. This row-column form sometimes is more useful than the whole table itself since we can enhance or filter each of its entries more easily.

For further research, this library can be extended with transformation / converter objects.

6.4 Display Library

A lot of computing power is spent on user display windowing systems nowadays. People can understand more easily what to do when they can see an icon/picture on the computer screen than when they just see a line of characters. However, programming using a graphical user interface is more demanding than programming with input / output being displayed on the regular TTY screen. Users need to know some of the basics of the

graphics library for the specific operating system. For instance, just to create a window on the X-Window system will involve a few initialization function calls.

The display library provides functions that create windows or a user display. The current library contains functions for creating a user display on the X-Window with the MOTIF user interface. The purpose of this function is to hide some of the complexity of the X data structures and function calls; users can use functions in this library to create a window and menus on it with little knowledge or even without knowledge of X-programming. The objective of this library is to allow users who do not know the X-Window programming system to create graphical components of their interface. However, there are possibilities for power users of X window to add or enhance the basic structure of the windows created with this library.

6.5 Miscellaneous Library

This library contains functions for general purposes. For instance, functiona for creating/inserting integer node linked list, or string node linked list. Also it is used for deleting node and releasing the memory use by the node in the linked list when the node is deleted from the list.

Chapter 7

Applications

This chapter contains some applications using functions created in this study. These programs are examples of how one can use the toolkit created in this study.

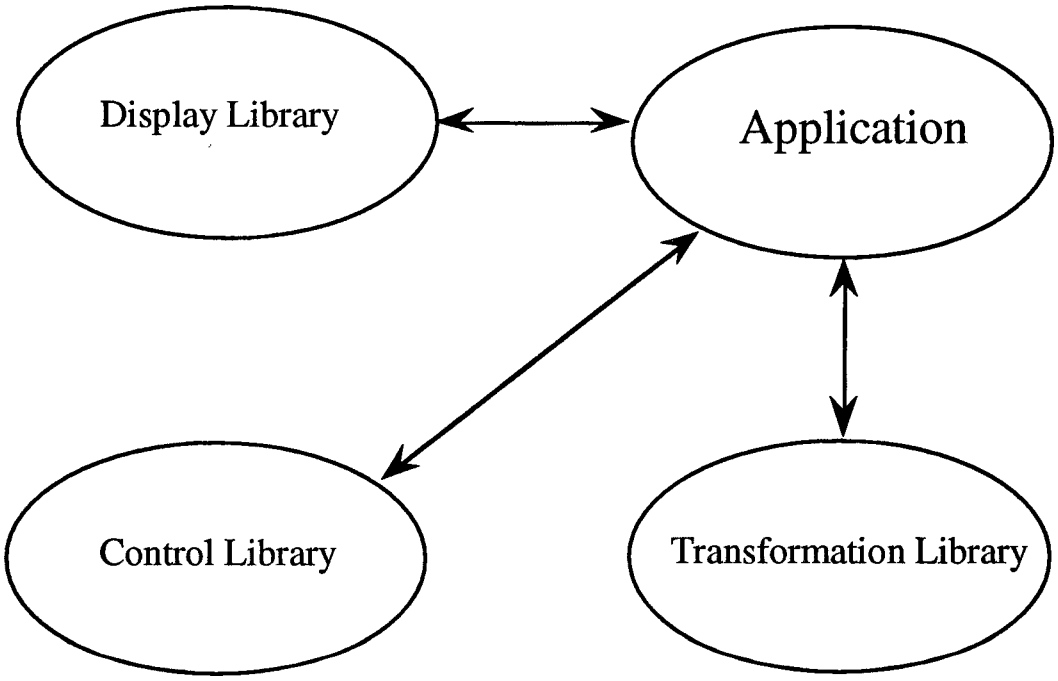

Fig 7-0 Application - libraries Relation

7.1 Monitoring process

The application consists of several windows: menubar, title, and display window (Fig. 7-1). The display window is a scroll window which can be scrolled vertically and horizontally.

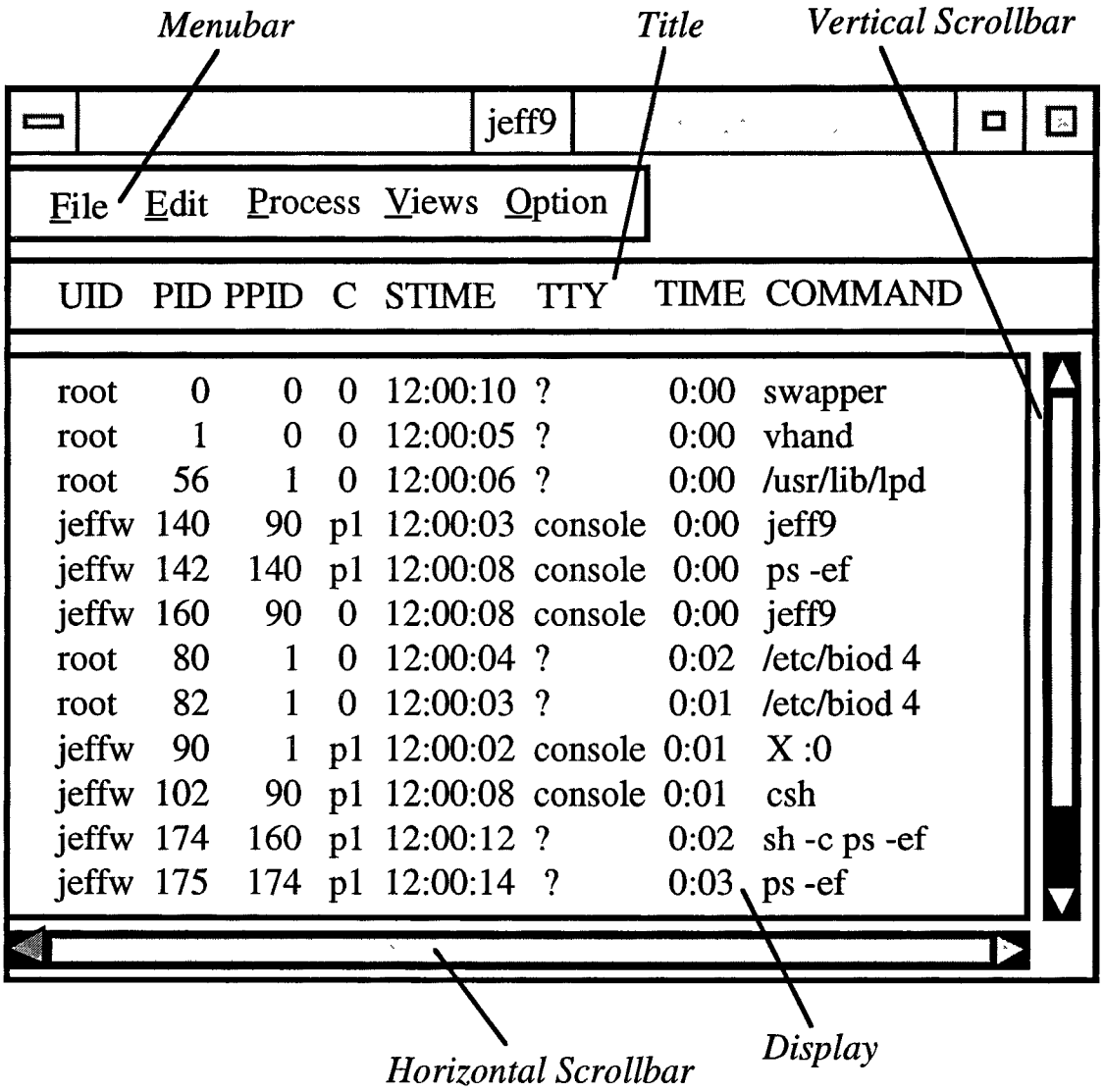

Fig. 7-1 Main Window

The window can be made larger or smaller by dragging any of the resize corners while pressing a mouse button. A user may select an item at the menu bar using a mouse or a keyboard. To select an item using the mouse, the user clicks on the desired item. To

select an item using the keyboard, the user types meta- underline. The meta key on an Apple keyboard is the command (cloverleaf) key, and it is used like a shift key. Every selection made on the menubar will display a pulldown menu. The user may select an item in the pulldown menu by clicking the item in this menu; a dialog window is displayed for every selection except for the 'quit' menu. The user may enter a new value in the appropriate dialog window and click the 'OK' button.

Saving to a file

The current display of the output screen can be saved to a file by clicking 'File' at the menubar or by pressing 'f' while holding down the meta-key, then clicking 'Save' at the pulldown menu. A pop-up dialog window will be displayed. The user may enter a file name at the blank window by clicking the window first (Fig. 7-2). After typing the file name, the user has an option to click the 'OK' button or 'CANCEL' button. Clicking the 'OK' button will save the file to the specified file name at the current directory and close the dialog window. Clicking the 'CANCEL' button will abort the operation and close the dialog window. The output of this screen will be saved as a regular text file.

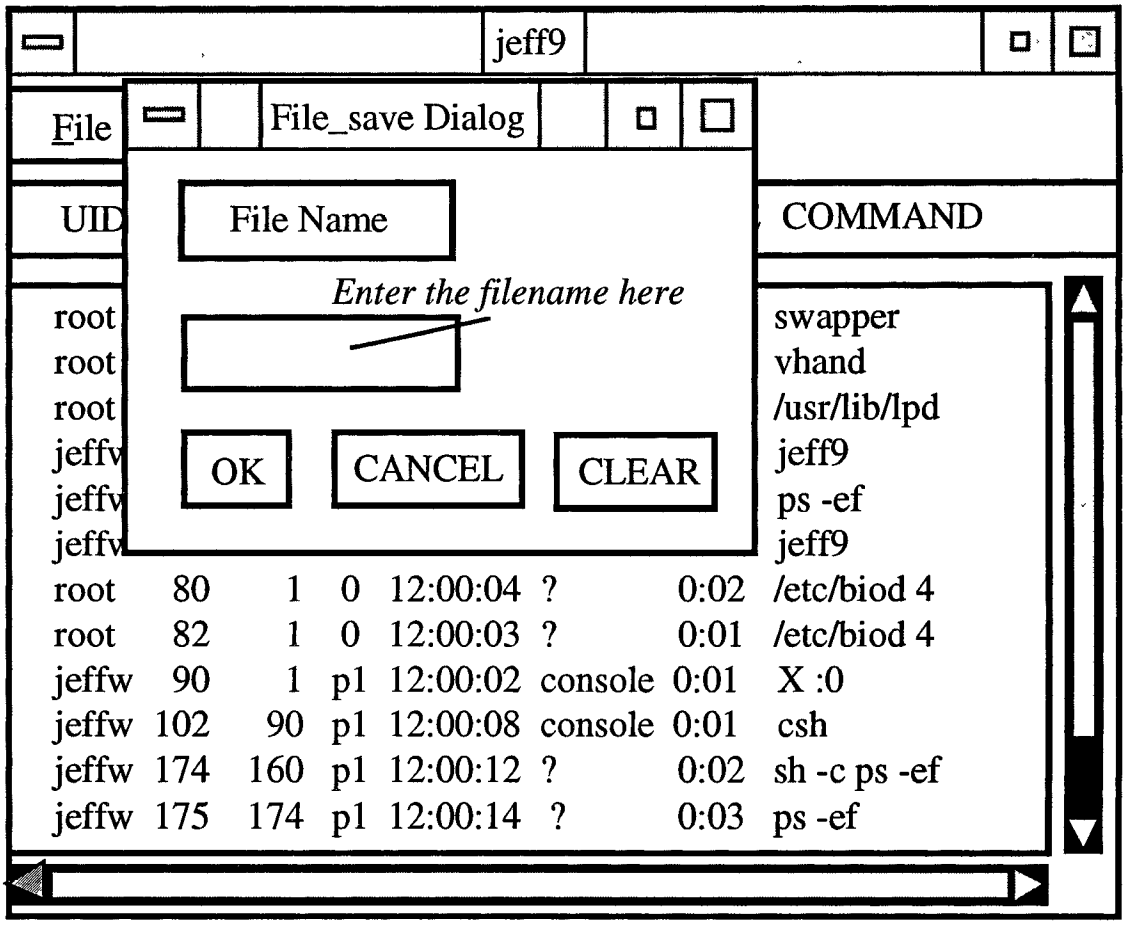

Fig. 7-2 File Save Pop-Up Window

Printing the current display to a printer

This application will print the current display to the default printer setup. Click the File menu and click the print menu on the File pulldown menu.

Quit The program

The only way to stop the application is from the 'Quit' menu under 'File' at the menubar. The close option that is usually available in the Motif Window Manager has been disabled, because the close in the MWM is only close display, while this application must also terminate the process.

Selecting a UNIX command

This application can take any one of the UNIX commands that writes output to standard output, preferably a UNIX command that creates a heading of the output. The default UNIX command is 'ps -ef'.

The current UNIX command can be changed by clicking 'Process' at the menubar or by pressing 'p' while holding down the meta-key, then clicking 'Command' at this pulldown menu. A pop-up dialog window will be displayed. The user may enter a UNIX command at the blank window by clicking the window first (Fig. 7-3). After typing the UNIX command, the user has an option to click the 'OK' button or the 'CANCEL' button. Clicking the 'OK' button will process the desired UNIX command and close the dialog window. Clicking the 'CANCEL' button will abort the operation and close the dialog window.

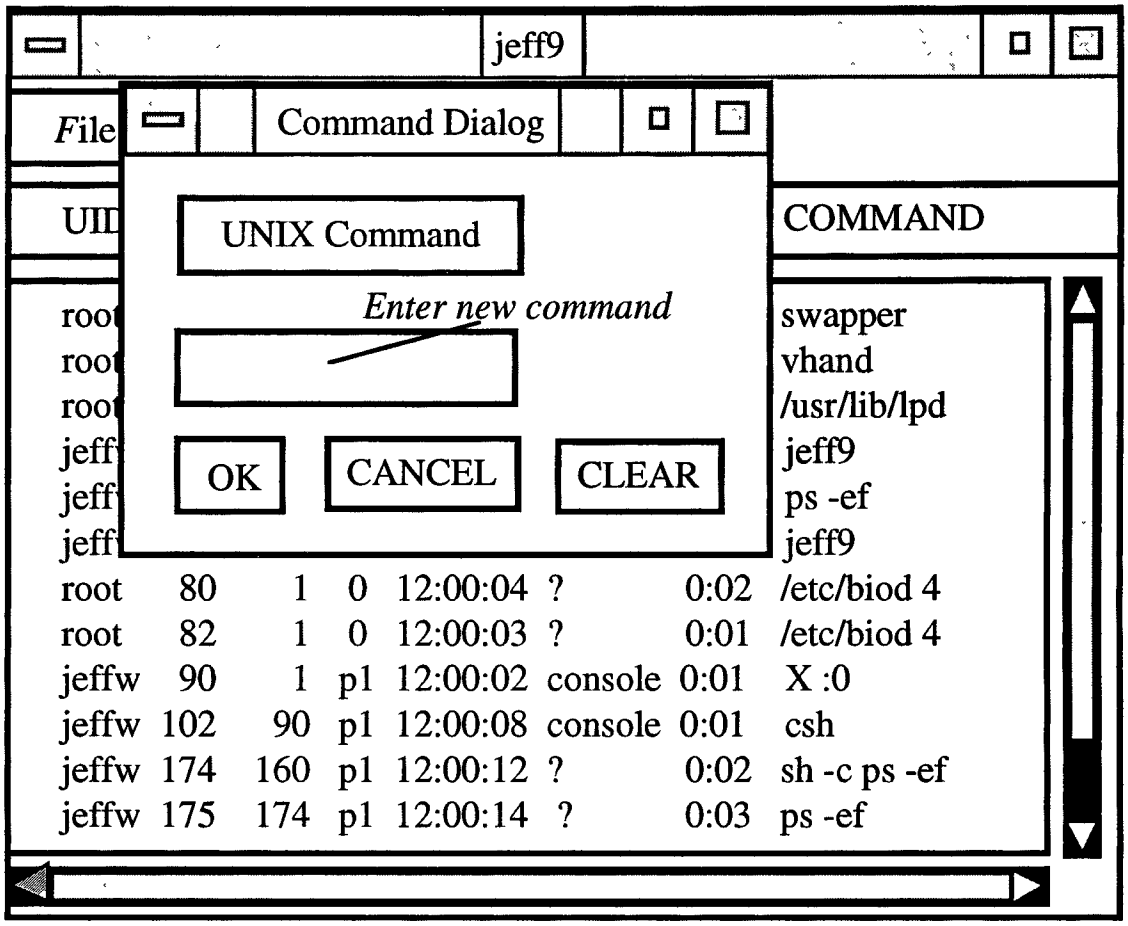

Fig. 7-3 Command Pop-Up Window

Changing the duration time

This application will process the given UNIX command in a loop with a certain delay time until the 'quit' menu is selected. The user may change the delay time by clicking 'Process' at the menubar or by pressing 'p' while holding down the meta-key, then clicking 'Delay' at this pulldown menu. A pop-up dialog window will be displayed. The user may enter a new delay time at the blank window by clicking the window first (Fig. 7-4). After typing a new delay time in units of seconds, the user has the option to click the 'OK' button or the 'CANCEL' button. Clicking the 'OK' button will process the desired delay time and close the dialog window. Clicking the 'CANCEL' button will abort the operation and close the dialog window.

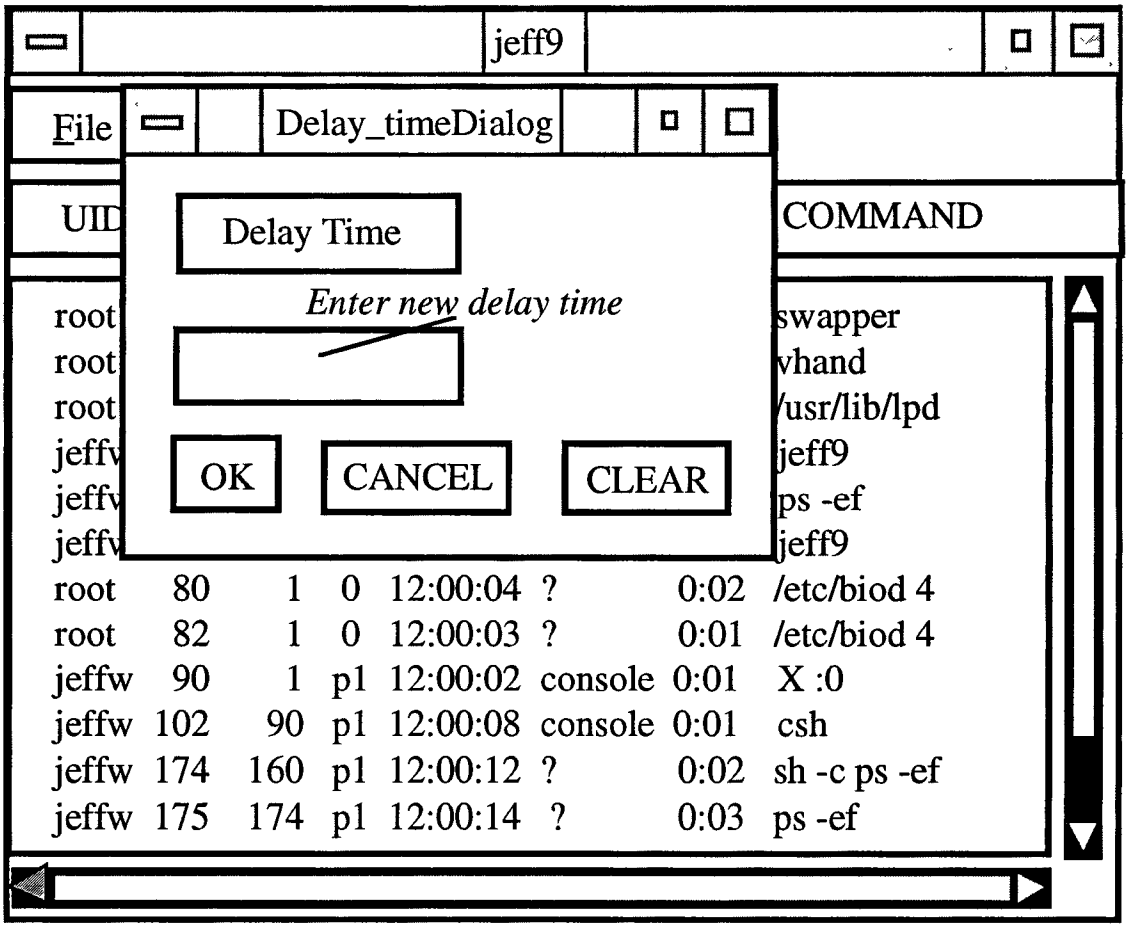

Fig. 7-4 Delay Pop-Up Window

Pause the process

This application allows the user to suspend the process. It will pause the process until the user selects the resume menu. Click 'Process' on the menubar and select the 'Pause' menu on the pulldown menu.

Resume the process

This application allows the user to resume the suspended process. The process will resume with the most current setup, including the changes that were made during the pause. Click the 'Process' on the menubar and select the 'Resume' menu on the pulldown menu.

Kill the selected process

This application allows the user to kill/terminate the selected process.

Selecting a Keyword for matching with a certain column

This application has the capability to display only the output that contains the matching keyword at a certain column. The default value for this keyword is blank which means it will match any string.

The current matching keyword can be changed by clicking 'Display' at the menubar or by pressing 'd' while holding down the meta-key, then clicking 'Key' at this pulldown menu. A pop-up dialog window will be displayed. The user may enter a keyword at the upper blank window and enter the column number at the lower blank window (Fig. 7-5). The user needs to click the mouse inside the window first before typing in the window. After typing the keyword and column number the user has an option to click the 'OK' button or 'CANCEL' button. Clicking the 'OK' button will process the desired keyword and close the dialog window. Clicking the 'CANCEL' button will abort the operation and close the dialog window.

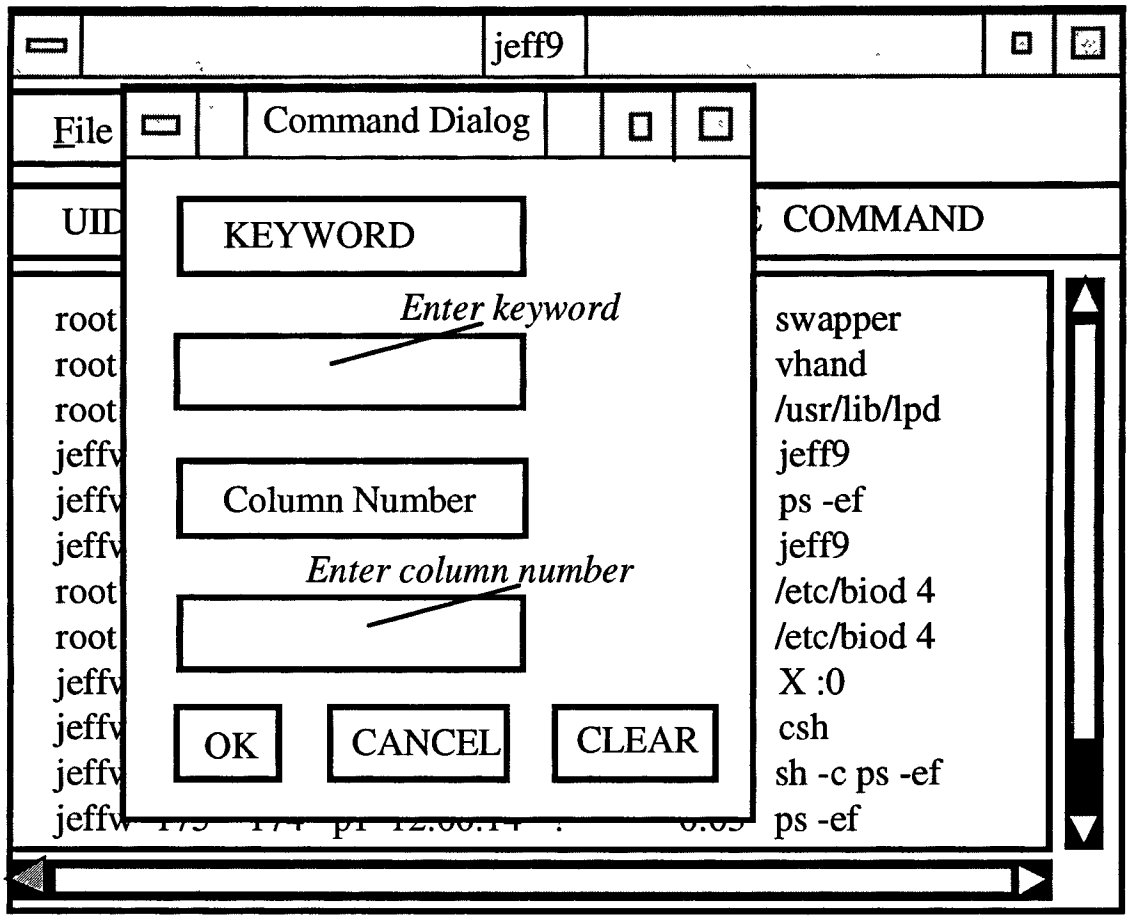

Fig. 7-5 Key Pop-Up Window

Column Selection

This application has a capability to display all columns or some selected columns. The default setting is to display all columns.

The current column selection can be changed by clicking 'Views' at the menubar or by pressing 'd' while holding down the meta-key, then clicking 'Column' at this pulldown menu. A pop-up dialog window will be displayed. The user may select the desired column by clicking the box next to the column name. The user has an option to click the 'OK' button or 'CANCEL' button. Clicking the 'OK' button will continue the process and display only those selected columns and then close the dialog window. Clicking the 'CANCEL' button will abort the operation and close the dialog window.

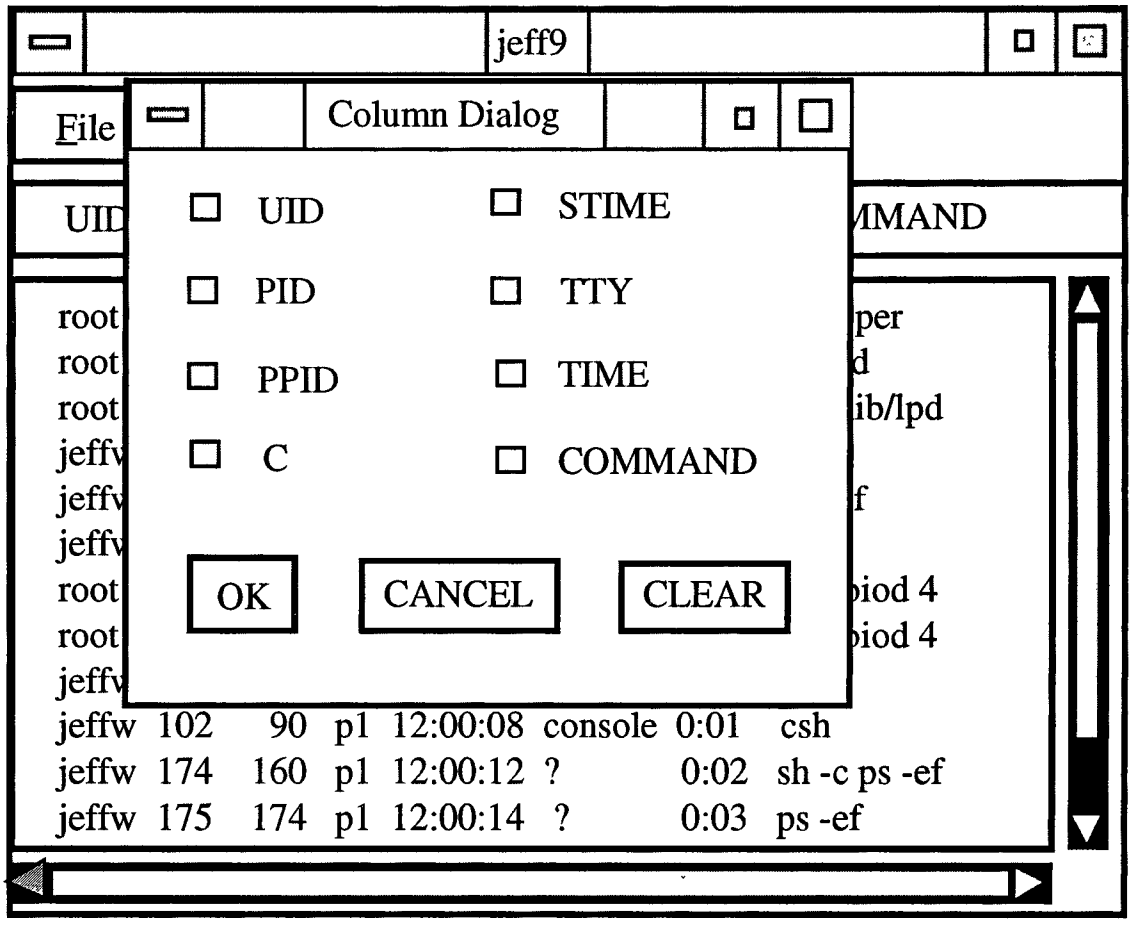

Fig. 7-6 Column Pop-Up Window

Row Selection

This application allows the user to display only selected rows. For instance, with the 'ps -ef' command the user might desire to monitor only processes belonging to root. The user can select the desired rows by clicking the desired rows and clicking 'Views' item on the menubar and selecting 'Row Display' on the pulldown menu.

Undo Row Selection

The user may cancel the selected row display only by selecting the 'Undo Row Selection' under the 'Views' item on the menubar. Undo Row Selection will remove the selected row from the selected item list and display all of the other rows again.

Snap Shot Option

This application allows the user to save the current display to another window. The new window runs independently of the main window; it remains open even if the main program/window is terminated (quit). Click the 'Option' on the menubar and select 'Snap Shot' on the pulldown menu.

7.2 Column Graph

This application consists of several windows: menubar, title, and display window. The display window is a graphic window which displays bar charts with respect to the output of the UNIX command that is processed in the background as another process. The option menus available for this application are:

- Quit menu under the File menu on the menubar is to terminate the application.
- Command menu under the Process menu on the menubar to change the background process.
- Title menu under the Process menu on the menubar to change the title desription on the title window.
- Duration menu under the Process menu on the menubar to change the delay time of the background process.
- Granulation menu under the Process menu on the menubar to change the number of partition graphs in the X direction.
- Scale menu under the Process menu on the menubar to change the height scale of the graph (the maximum number of a bar on the graph window).

7.3 Text Display

This application consists of several windows: menubar, title, and display window. The display window is a graphic window which displays text that is entered in the command line argument when running the program. The option menus available for this application are:

- Save menu under the File menu on the menubar to save the contents of the display to a file.
- Print menu under the File menu on the menubar to print the contents of the display to a standard printer output.
- Quit menu under the File menu on the menubar to terminate the application.

Chapter 8

Conclusions and Further Research

In this study we explored the possibility of a new toolkit or expressive language that can be used by programmers. We found out there are needs for:

- Object oriented encapsulation of code and data hiding to support a reusable module and minimize the amount of time for programmers to learn this new toolkit or language.
- Network transparent data sharing so the programmer doesn't have to understand communication details, such as the data structure required when creating connections between processes or/and machines.
- Graphical user interface with graphical specifications independent of a particular standard.
- Conversion objects to convert one form of data to another form, for example: encryption and decryption, compression and decompression.

During the study we created a new Toolkit which consists of: a communication library which consists of communication functions for transfering data between processes on the same machine, a control library which consists of functions that monitor processes, a tranformation library which consists of functions for converting a table form into row-column form and vice-versa, and a graphical user interface library which consists of functions for creating display windows, menubar, menu item on the X-Window MOTIF user interface.

The toolkit can be enhanced and expanded, so it might become a language in its own right. This language would support features such as object-oriented design, graphical user interface, network transparency, fault tolerance, conversion of objects, and ease of use.

The toolkit created in this research is using a UNIX operating system and X-Window graphical user interface, the tollkit can be enhanced by providing functions that will work on other operating system (such as, VAX-VMS, MS_DOS, OS/2, and mac operating system) and different graphical user interface (such as, OPEN LOOK, MS-Window, and Macintosh Window).

Bibliography

1. Bacon, Ian. "Opening the X Window," Mac User, June 1992, 205-210.

2. Booch, Grady. Software Component with ADA. Menlo Park, CA: The Benjamin/ Cummings Publishing Company Inc., 1987.

3. Brown, Judith R. and Cunningham, Steve. Programming the User Interface Principles and Examples. New York, NY: John Wiley &Sons, Inc. 1989.

4. Goodwin, Mark. User Interface in C: Programmer's Guide to State-of-The-Art Interfaces. Portland, OR: MIS Press, 1989.

5. Heller, Dan. Motif Programming Manual. OSF/Motif Edition, The definitive Guides to the X Window System Volume Six, Sebastopol, CA: O'Reilly & associates, Inc., 1991.

6. Hooper, James W. and Rowena O. Chester. Software Reuse: Guidelines and Methods. New York, NY: Plenum Press, 1991.

7. Johnson, Eric F. and Kevin Reichard. Advance X Window Applications Programming: The Basics and Beyond. Portland, OR: MIS Press, 1990.

8. Johnson, Eric F. and Kevin Reichard. Power Programming MOTIF. Portland, OR: MIS Press, 1991.

9. Johnson, Eric F. and Kevin Reichard. X Window Applications Programming. Portland, OR: MIS Press, 1989.

10. Lamb, David A. Software Engineering: Planning for Change. Englewood Cliffs, N.J.: Prentice Hall, 1988.

11. Mullender, Sape. Distributed Systems. New York, NY :ACM Press, 1989.

12. Newman, Ron, James Gettys, and Robert Scheifler, X Window System: C Library and Protocol Reference. Digital Press, 1988.

13. Nye, Adrian and Tim O'Reilly. X Toolkit Intrinsics Programming Manual, OSF/Motif Edition, The definitive Guides to the X Window System Volume Four, Sebastopol, CA: O'Reilly & associates, Inc. 1990.

14. Nye, Adrian and Tim O'Reilly. X lib Programming Manual, The definitive Guides to the X Window System Volume One, Sebastopol, CA: O'Reilly & associates, Inc. 1990.

15. Open Software Foundation. OSF/Motif: Programmer's Guide, Revision 1.1. Englewood Cliffs, N.J.: Prentice Hall 1991.

16. Open Software Foundation. OSF/Motif: Programmer's Reference, Revision 1.1. Englewood Cliffs, N.J.: Prentice Hall 1991.

17. The definitive Guides to the X Window System Volume Three, Sebastopol, CA: O'Reilly & associates, Inc. 1990.

18. Rochkind, Marc J. Advance UNIX Programming. Englewood Cliffs, N.J.: Prentice Hall, 1985.

19. Stallman, Richard and Simson Garfinkel. "Against User Interface Copyright" Communications of The ACM Nov. 1990: 15-18.

20. Young, Douglas A. The X Window System Programming and Applications with Xt, OSF/MOTIF Edition. Englewood, N.J.: Prentice Hall, 1990.

21. Young, Douglas A. and John A. Pew The X Window System Programming and Applications with Xt, OPEN LOOK Edition. Englewood, N.J.: Prentice Hall, 1992.

Appendix A

Applications' code

A.1 main.c

```
/*************************************************************/
/* program     : main.c                                      */
/* programmer  : Jeffrey Wijono                              */
/* description : Example program of using Jeff Library       */
/*               The program will transform an Unix Command into rows */
/*               and columns, that the control library will add or */
/*               enhance these rows and columns, and using the display */
/*               library it displays the output into a Motif Windows */
/*************************************************************/

#include <stdio.h>
#include "miscellaneous.h"
#include "display.h"
#include "control.h"
#include "trans.h"

void  user_initialize();
void  user_create_menu();

main(argc, argv)
 int argc;
 char *argv[];
{

  user_initialize();

  /* spawn the process for monitoring the command */

  monitor(&argc, argv);

  /* spawn the process for creating the display window */

  create_list_display(argc, argv);

  /* transform the output of monitor into row col linked list and
     display it into the display window */

  user_loop();
}


void user_initialize()

{
  initial_control_lib();
  initial_list_display();
```

```
  set_process_command("ps -ef | grep -v defunct");
  set_delay_time(DELAY_TIME);
  user_create_menu();
}


void user_create_menu()

{
    initialize_menu();

    menu("File", NULL);
    menu_item("Save", DoSaveList);
    menu_item("Print", DoPrintList);
    menu_item("Quit", DoQuit);
    done_menu();

    menu("Process", NULL);
    menu_item("Command", DoCommand);
    menu_item("Duration", DoDelay);
    menu("SUB 1", NULL);
    menu_item("Pause",DoPause);
    menu_item("Resume", DoResume);
    done_menu();
    menu_item("Kill Process", DoKillProcess);
    done_menu();

    menu("View", NULL);
    menu_item("Key     Select", DoKeySelect);
    menu_item("Column  Select", DoColSelect);
    menu_item("Row     Select", DoRowSelect);
    menu_item("Undo Row Select", UndoRowSelect);
    done_menu();

    menu("Option", NULL);
    menu_item("Snap Shot", DoSnap);
    done_menu();
}

user_loop()

{
int cols[MAX_COLUMN+1], col_of_key, i;
char buf[COL_WIDTH], str[COL_WIDTH], key[40];
node_struct *head, *head2, *head3;
row_col_struct *row_col_head;
pid_list_struct *pid_list_head, *cur_pid_list;
int_node *pid_head, *select_head;

    pid_list_head = NULL;
    select_head = NULL;
    row_col_head = NULL;
```

```
for (i=0; i<=MAX_COLUMN; i++)
    cols[i] = i+1;
buf[0] = '\0';
key[0] = '\0';
col_of_key = 0;
for ( ; ; ) {
/* create a linked list of rows from the output of monitor */
    receive_from_control(&head);
    if (head != NULL) {
    row_col_head = trans_to_row_col(head);

/* transfer row_col_head list to a list to be displayed */
    head2 = create_display_list(row_col_head, key, cols, col_of_key,
                                    &pid_list_head, select_head);

/* send the final list to display window */
    send_data_to_display(head2);

/* clear all the list */
    clear_node(&head);
    clear_node(&head2);
    clear_row_col_list(&row_col_head);
    }
/* read if there is any request from the menu of the display window */
    receive_from_display(&head3);
    if (head3) {
        if (is_command(head3, "_TERMINATE") == 0) {
            send_to_control(head3);
            send_command_to_display(head3);
            exit(0);
            }
        else if (is_command(head3, "_CHANGE_COMMAND") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_CHANGE_DELAY") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_CHANGE_KEY") == 0) {
            if (head3->next != NULL) {
                strcpy(key, head3->next->line);
                if (head3->next->next != NULL)
                    col_of_key = atoi (head3->next->next->line);
                else
                    col_of_key = 0;
                }
            else {
                strcpy(key, "*");
                col_of_key = 0;
                }
            }
        else if (is_command(head3, "_PAUSE") == 0) {
            send_to_control(head3);
```

```
            }
        else if (is_command(head3, "_RESUME") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_KILL_PROCESS") == 0) {
            do_kill_process(head3, pid_list_head);
            }
        else if (is_command(head3, "_ROW_SELECT") == 0) {
            clear_int_node(&select_head);
            do_row_selection(head3, pid_list_head, &select_head);
            }
        else if (is_command(head3, "_UNDO_ROW_SELECT") == 0) {
            clear_int_node(&select_head);
            }
        else if (is_command(head3, "_COL_SELECT") == 0) {
            strcpy(str, head3->next->line);
            str_to_array(str, cols);
            }
        else if (is_command(head3, "_CLEAR_LIST_DISPLAY") == 0) {
        /* clear the first pid_head in the pid_list_head */
            if (pid_list_head != NULL) {
                cur_pid_list = pid_list_head;
                pid_list_head = pid_list_head->next;
                pid_head = cur_pid_list->list;
                clear_int_node(&pid_head);
                free(cur_pid_list);
                }
            }
        clear_node(&head3);
        }
    }   /* end for loop     */
}       /* end of user_loop  */


do_row_selection(head, pid_list_head, select_head)
node_struct *head;
pid_list_struct *pid_list_head;
int_node **select_head;

{
node_struct *cur;
int pid;
int_node *pid_head2;

    pid_head2 = NULL;
    if (pid_list_head != NULL)
    for (cur = head->next; cur != NULL; cur = cur->next) {
        pid = row_get_pid(pid_list_head, cur->line);
        if (pid >= 0)
            insert_int_node(&pid_head2, pid);
        }
    *select_head = pid_head2;
```

```
}   /* do_row_selection */


do_kill_process(head, pid_list_head)
node_struct *head;
pid_list_struct *pid_list_head;

{
node_struct *cur;
char kill_str[20];
int pid;

    if (pid_list_head != NULL)
    for (cur = head->next; cur != NULL; cur = cur->next) {
        pid = row_get_pid(pid_list_head, cur->line);
        if (pid > 0) {
            sprintf(kill_str, "kill %d", pid);
            system(kill_str);
            }
        }
}       /* do_kill_process */


row_get_pid(pid_list_head, str)
pid_list_struct *pid_list_head;
char *str;

{
int_node *cur_pid;
int i, row;

    row = atoi(str);
    for (i=0, cur_pid = pid_list_head->list; cur_pid != NULL;
            cur_pid = cur_pid->next, i++) {
        if (row == i)
            return(cur_pid->num);
        }   /* for cur_pid != NULL */
    return(0);
}   /* row_get_pid */
```

A.2 main2.c

```
/*************************************************************/
/* program      : main.c                                     */
/* programmer   : Jeffrey Wijono                             */
/* description  : Example program of using Jeff Library      */
/*                The program will transform an Unix Command into rows */
/*                and columns, that the control library will add or */
/*                enhance these rows and columns, and using the display */
/*                library it displays the output into a Motif Windows */
/*************************************************************/
```

```
#include <stdio.h>
#include "miscellaneous.h"
#include "display.h"
#include "control.h"
#include "trans.h"

void  user_initialize();
void  user_create_menu();

main(argc, argv)
  int argc;
  char *argv[];
{

   user_initialize();

   /* spawn the process for monitoring the command */

   monitor_message(&argc, argv);

   /* spawn the process for creating the display window */

   create_list_display(argc, argv);

   /* transform the output of monitor into row col linked list and
      display it into the display window */

   user_loop();
}


void user_initialize()

{
   initial_control_lib();
   initial_list_display();
   set_process_command("ps -ef | grep -v defunct");
   set_delay_time(DELAY_TIME);
   user_create_menu();
}


void user_create_menu()

{
      initialize_menu();

      menu("File", NULL);
      menu_item("Save", DoSaveList);
      menu_item("Print", DoPrintList);
      menu_item("Quit", DoQuit);
```

```
    done_menu();

    menu("Process", NULL);
    menu_item("Command", DoCommand);
    menu_item("Duration", DoDelay);
    menu("SUB 1", NULL);
    menu_item("Pause",DoPause);
    menu_item("Resume", DoResume);
    done_menu();
    menu_item("Kill Process", DoKillProcess);
    done_menu();

    menu("View", NULL);
    menu_item("Key     Select", DoKeySelect);
    menu_item("Column  Select", DoColSelect);
    menu_item("Row     Select", DoRowSelect);
    menu_item("Undo Row Select", UndoRowSelect);
    done_menu();

    menu("Option", NULL);
    menu_item("Snap Shot", DoSnap);
    done_menu();
}

user_loop()

{
int cols[MAX_COLUMN+1], col_of_key, i;
char buf[COL_WIDTH], str[COL_WIDTH], key[40];
node_struct *head, *head2, *head3;
row_col_struct *row_col_head;
pid_list_struct *pid_list_head, *cur_pid_list;
int_node *pid_head, *select_head;

    pid_list_head = NULL;
    select_head = NULL;
    row_col_head = NULL;
    for (i=0; i<=MAX_COLUMN; i++)
        cols[i] = i+1;
    buf[0] = '\0';
    key[0] = '\0';
    col_of_key = 0;
    for ( ; ; ) {
    /* create a linked list of rows from the output of monitor */
        receive_from_control(&head);
        if (head != NULL) {
        row_col_head = trans_to_row_col(head);

    /* transfer row_col_head list to a list to be displayed */
        head2 = create_display_list(row_col_head, key, cols, col_of_key,
                                    &pid_list_head, select_head);
```

```
/* send the final list to display window */
    send_data_to_display(head2);

/* clear all the list */
    clear_node(&head);
    clear_node(&head2);
    clear_row_col_list(&row_col_head);
    }
/* read if there is any request from the menu of the display window */
    receive_from_display(&head3);
    if (head3) {
        if (is_command(head3, "_TERMINATE") == 0) {
            send_to_control(head3);
            send_command_to_display(head3);
            exit(0);
            }
        else if (is_command(head3, "_CHANGE_COMMAND") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_CHANGE_DELAY") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_CHANGE_KEY") == 0) {
            if (head3->next != NULL) {
                strcpy(key, head3->next->line);
                if (head3->next->next != NULL)
                    col_of_key = atoi (head3->next->next->line);
                else
                    col_of_key = 0;
                }
            else {
                strcpy(key, "*");
                col_of_key = 0;
                }
            }
        else if (is_command(head3, "_PAUSE") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_RESUME") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_KILL_PROCESS") == 0) {
            do_kill_process(head3, pid_list_head);
            }
        else if (is_command(head3, "_ROW_SELECT") == 0) {
            clear_int_node(&select_head);
            do_row_selection(head3, pid_list_head, &select_head);
            }
        else if (is_command(head3, "_UNDO_ROW_SELECT") == 0) {
            clear_int_node(&select_head);
            }
        else if (is_command(head3, "_COL_SELECT") == 0) {
```

```
                strcpy(str, head3->next->line);
                str_to_array(str, cols);
                }
            else if (is_command(head3, "_CLEAR_LIST_DISPLAY") == 0) {
            /* clear the first pid_head in the pid_list_head */
                if (pid_list_head != NULL) {
                    cur_pid_list = pid_list_head;
                    pid_list_head = pid_list_head->next;
                    pid_head = cur_pid_list->list;
                    clear_int_node(&pid_head);
                    free(cur_pid_list);
                    }
                }
            clear_node(&head3);
            }
        }   /* end for loop     */
}       /* end of user_loop  */


do_row_selection(head, pid_list_head, select_head)
node_struct *head;
pid_list_struct *pid_list_head;
int_node **select_head;

{
node_struct *cur;
int pid;
int_node *pid_head2;

    pid_head2 = NULL;
    if (pid_list_head != NULL)
    for (cur = head->next; cur != NULL; cur = cur->next) {
        pid = row_get_pid(pid_list_head, cur->line);
        if (pid >= 0)
            insert_int_node(&pid_head2, pid);
        }
    *select_head = pid_head2;
}   /* do_row_selection */


do_kill_process(head, pid_list_head)
node_struct *head;
pid_list_struct *pid_list_head;

{
node_struct *cur;
char kill_str[20];
int pid;

    if (pid_list_head != NULL)
    for (cur = head->next; cur != NULL; cur = cur->next) {
        pid = row_get_pid(pid_list_head, cur->line);
```

```
        if (pid > 0) {
            sprintf(kill_str, "kill %d", pid);
            system(kill_str);
            }
        }
}       /* do_kill_process */


row_get_pid(pid_list_head, str)
pid_list_struct *pid_list_head;
char *str;

{
int_node *cur_pid;
int i, row;

    row = atoi(str);
    for (i=0, cur_pid = pid_list_head->list; cur_pid != NULL;
            cur_pid = cur_pid->next, i++) {
        if (row == i)
            return(cur_pid->num);
        }   /* for cur_pid != NULL */
    return(0);
}   /* row_get_pid */
```

A.3 main3.c

```
/*************************************************************/
/*  program      : main.c                                    */
/*  programmer   : Jeffrey Wijono                            */
/*  description  : Example program of using Jeff Library     */
/*                 The program will transform an Unix Command into rows */
/*                 and columns, that the control library will add or */
/*                 enhance these rows and columns, and using the display */
/*                 library it displays the output into a Motif Windows */
/*************************************************************/

#include <stdio.h>
#include "miscellaneous.h"
#include "display.h"
#include "control.h"
#include "trans.h"

void  user_initialize();
void  user_create_menu();

main(argc, argv)
 int argc;
 char *argv[];
{
```

```
  user_initialize();

  /* spawn the process for monitoring the command */

  monitor_message(&argc, argv);

  /* spawn the process for creating the display window */

  create_list_display_message(argc, argv);

  /* transform the output of monitor into row col linked list and
     display it into the display window */

  user_loop();
}


void user_initialize()

{
  initial_control_lib();
  initial_list_display();
  set_process_command("ps -ef | grep -v defunct");
  set_delay_time(DELAY_TIME);
  user_create_menu();
}


void user_create_menu()

{
    initialize_menu();

    menu("File", NULL);
    menu_item("Save", DoSaveList);
    menu_item("Print", DoPrintList);
    menu_item("Quit", DoQuit);
    done_menu();

    menu("Process", NULL);
    menu_item("Command", DoCommand);
    menu_item("Duration", DoDelay);
    menu("SUB 1", NULL);
    menu_item("Pause",DoPause);
    menu_item("Resume", DoResume);
    done_menu();
    menu_item("Kill Process", DoKillProcess);
    done_menu();

    menu("View", NULL);
    menu_item("Key     Select", DoKeySelect);
    menu_item("Column  Select", DoColSelect);
```

```
    menu_item("Row     Select", DoRowSelect);
    menu_item("Undo Row Select", UndoRowSelect);
    done_menu();

    menu("Option", NULL);
    menu_item("Snap Shot", DoSnap);
    done_menu();
}

user_loop()

{
int cols[MAX_COLUMN+1], col_of_key, i;
char buf[COL_WIDTH], str[COL_WIDTH], key[40];
node_struct *head, *head2, *head3;
row_col_struct *row_col_head;
pid_list_struct *pid_list_head, *cur_pid_list;
int_node *pid_head, *select_head;

    pid_list_head = NULL;
    select_head = NULL;
    row_col_head = NULL;
    for (i=0; i<=MAX_COLUMN; i++)
        cols[i] = i+1;
    buf[0] = '\0';
    key[0] = '\0';
    col_of_key = 0;
    for ( ; ; ) {
    /* create a linked list of rows from the output of monitor */
        receive_from_control(&head);
        if (head != NULL) {
        row_col_head = trans_to_row_col(head);

    /* transfer row_col_head list to a list to be displayed */
        head2 = create_display_list(row_col_head, key, cols, col_of_key,
                                    &pid_list_head, select_head);

    /* send the final list to display window */
        send_data_to_display(head2);

    /* clear all the list */
        clear_node(&head);
        clear_node(&head2);
        clear_row_col_list(&row_col_head);
        }
    /* read if there is any request from the menu of the display window */
        receive_from_display(&head3);
        if (head3) {
            if (is_command(head3, "_TERMINATE") == 0) {
                send_to_control(head3);
                send_command_to_display(head3);
                exit(0);
```

```
    }
else if (is_command(head3, "_CHANGE_COMMAND") == 0) {
    send_to_control(head3);
    }
else if (is_command(head3, "_CHANGE_DELAY") == 0) {
    send_to_control(head3);
    }
else if (is_command(head3, "_CHANGE_KEY") == 0) {
    if (head3->next != NULL) {
        strcpy(key, head3->next->line);
        if (head3->next->next != NULL)
            col_of_key = atoi (head3->next->next->line);
        else
            col_of_key = 0;
        }
    else {
        strcpy(key, "*");
        col_of_key = 0;
        }
    }
else if (is_command(head3, "_PAUSE") == 0) {
    send_to_control(head3);
    }
else if (is_command(head3, "_RESUME") == 0) {
    send_to_control(head3);
    }
else if (is_command(head3, "_KILL_PROCESS") == 0) {
    do_kill_process(head3, pid_list_head);
    }
else if (is_command(head3, "_ROW_SELECT") == 0) {
    clear_int_node(&select_head);
    do_row_selection(head3, pid_list_head, &select_head);
    }
else if (is_command(head3, "_UNDO_ROW_SELECT") == 0) {
    clear_int_node(&select_head);
    }
else if (is_command(head3, "_COL_SELECT") == 0) {
    strcpy(str, head3->next->line);
    str_to_array(str, cols);
    }
else if (is_command(head3, "_CLEAR_LIST_DISPLAY") == 0) {
/* clear the first pid_head in the pid_list_head */
    if (pid_list_head != NULL) {
        cur_pid_list = pid_list_head;
        pid_list_head = pid_list_head->next;
        pid_head = cur_pid_list->list;
        clear_int_node(&pid_head);
        free(cur_pid_list);
        }
    }
clear_node(&head3);
}
```

```
        }    /* end for loop      */
}       /* end of user_loop  */


do_row_selection(head, pid_list_head, select_head)
node_struct *head;
pid_list_struct *pid_list_head;
int_node **select_head;

{
node_struct *cur;
int pid;
int_node *pid_head2;

    pid_head2 = NULL;
    if (pid_list_head != NULL)
    for (cur = head->next; cur != NULL; cur = cur->next) {
        pid = row_get_pid(pid_list_head, cur->line);
        if (pid >= 0)
            insert_int_node(&pid_head2, pid);
        }
    *select_head = pid_head2;
}   /* do_row_selection */


do_kill_process(head, pid_list_head)
node_struct *head;
pid_list_struct *pid_list_head;

{
node_struct *cur;
char kill_str[20];
int pid;

    if (pid_list_head != NULL)
    for (cur = head->next; cur != NULL; cur = cur->next) {
        pid = row_get_pid(pid_list_head, cur->line);
        if (pid > 0) {
            sprintf(kill_str, "kill %d", pid);
            system(kill_str);
            }
        }
}       /* do_kill_process */


row_get_pid(pid_list_head, str)
pid_list_struct *pid_list_head;
char *str;

{
int_node *cur_pid;
int i, row;
```

```
    row = atoi(str);
    for (i=0, cur_pid = pid_list_head->list; cur_pid != NULL;
            cur_pid = cur_pid->next, i++) {
        if (row == i)
            return(cur_pid->num);
        }   /* for cur_pid != NULL */
    return(0);
}   /* row_get_pid */
```

A.4 graph.c

```
/*****************************************************************/
/*  program      : main.c                                        */
/*  programmer   : Jeffrey Wijono                                */
/*  description  : Example program of using Jeff Library         */
/*                 The program will accept a number and display it as a */
/*                 bar graph on the output display window        */
/*****************************************************************/

#include <stdio.h>
#include "miscellaneous.h"
#include "display.h"
#include "trans.h"
#include "control.h"

void  user_initialize();
void  user_create_menu();

main(argc, argv)
int argc;
char *argv[];
{

    user_initialize();

    /* spawn the process for monitoring the command */

    monitor(&argc, argv);

    /* spawn the process for creating the display window */

    create_graph_display(argc, argv);

    /* transform the output of monitor into row col linked list and
        display it into the display window */

    user_loop();
}

void user_initialize()
```

```
{
    initial_control_lib();
    initial_graph_display();
    set_process_command("ps -ef | wc -l");
    set_delay_time(DELAY_TIME);
    user_create_menu();
}

void user_create_menu()

{
    initialize_menu();

    menu("File", NULL);
    menu_item("Quit", DoQuit);
    done_menu();

    menu("Process", NULL);
    menu_item("Command ", DoCommand);
    menu_item("Title", DoSetTitle);
    menu_item("Duration", DoDelay);
    menu_item("Granulation", DoGranulation);
    menu_item("Scale     ", DoScale);
    done_menu();

}

user_loop()

{
int cols[MAX_COLUMN+1], col_of_key, i;
char buf[COL_WIDTH], str[COL_WIDTH], key[40];
node_struct *head, *head2, *head3;

    buf[0] = '\0';
    key[0] = '\0';
    for ( ; ; ) {
    /* create a linked list of rows from the output of monitor */
    receive_from_control(&head);
    if (head) {
    /* send the final list to display window */
        send_data_to_display(head);

    /* clear all the list */
        clear_node(&head);
        }

    /* read if there is any request from the menu of the display window */
    receive_from_display(&head3);
    if (head3) {
        if (is_command(head3, "_TERMINATE") == 0) {
```

```
            send_to_control(head3);
            send_command_to_display(head3);
            exit(0);
            }
        else if (is_command(head3, "_CHANGE_COMMAND") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_CHANGE_DELAY") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_PAUSE") == 0) {
            send_to_control(head3);
            }
        else if (is_command(head3, "_RESUME") == 0) {
            send_to_control(head3);
            }
        clear_node(&head3);
        }
    }   /* end for loop     */
}       /* end of user_loop  */
```

A.5 snap.c

```
/*************************************************************/
/*  program      : snap.c                                    */
/*  programmer   : Jeffrey Wijono                            */
/*  description  : Sample of using Text Widget               */
/*************************************************************/

#include <stdio.h>
#include "miscellaneous.h"
#include "display.h"
#include "control.h"


main(argc, argv)
int argc;
char *argv[];

{
    user_initialize();

    create_text_display(argc, argv);

/* text_snap(argc, argv); */
}


user_initialize()
```

```
{
    initial_text_display();
    user_create_menu();
}   /* user_initialize */


user_create_menu()

{
    initialize_text_menu();

    menu("File", NULL);
    menu_item("Save", DoSaveText);
    menu_item("Print", DoPrintText);
    menu_item("Quit", DoQuit2);
    done_menu();
}   /* user_cretae_menu */
```

Appendix B

Communication Library's code

B.1 communication.h

```
/*************************************************************/
/* Program      : communication.h                             */
/* Programmer   : Jeffrey Wijono                              */
/* description  : This file contains header file for communication library */
/*************************************************************/

typedef struct _shared_mem_pipe_struct {
        char    *addr;          /* shared memory block address  */
        int     semid;          /* semaphore key  */
        int     segid;          /* segment ID */
        } shared_mem_pipe;

int     (*read_function)();
int     *write_function)();
int     read_shared_mem();
int     write_shared_mem();
int     read_message();
int     write_message();
int     read();
int     write();
int     close_pipe();
int     close_shared_mem_pipe();
int     close_message_pipe();
```

B.2 communication.c

```
/*************************************************************/
/* Program      : communication.c                             */
/* Programmer   : Jeffrey Wijono                              */
/* description  : This file contains functions used for communication */
/*                between processes.                          */
/*************************************************************/


#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include "communication.h"
#include "miscellaneous.h"

#define BUFSIZE 512
```

```
/********************** shared memory pipe *********************/

void    P();
void    V();


static void semcall(sid, op)
int sid, op;

{
struct sembuf sb;

    sb.sem_num = 0;
    sb.sem_op = op;
    sb.sem_flg = 0;
    if (semop(sid, &sb, 1) == -1)
        syserr("semop");
}   /*  semcall  */


void P(sid)
int sid;

{
    semcall(sid, -1);
}   /* P */


void V(sid)
int sid;

{
    semcall(sid, 1);
}   /* V */


create_shared_mem_pipe(fd_initial)
int *fd_initial;
{
int *tmp;
shared_mem_pipe *fd;


    if ((fd = (shared_mem_pipe *) malloc(sizeof(shared_mem_pipe))) == NULL)
        syserr("malloc");
    fd_initial[0] = fd_initial[1] = (int) fd;
    if((fd->segid = shmget(IPC_PRIVATE, BUFSIZE, 0666)) == -1)
        syserr("shmget");
    if ((int) (fd->addr = (char *) shmat(fd->segid, 0, 0)) == -1)
        syserr("shmat");
    if ((fd->semid = semget((key_t) fd->segid, 1, 0666 | IPC_CREAT)) == -1)
        syserr("semget");
```

```
    V(fd->semid);
    tmp = (int *)(fd->addr);
    *tmp = *(tmp + 1) = (int)(tmp + 2);
    return ((int) fd);
    sleep(2);
}   /* create_shared_mem_pipe */


close_shared_mem_pipe(fds)
int fds;

{
    shared_mem_pipe *fd;
    fd = (shared_mem_pipe *) fds;

    if (semctl(fd->semid, 0, IPC_RMID, 0) == -1) syserr("semctl");
    if (shmdt(fd->addr) == -1) syserr("shmdt");
    if (shmctl(fd->segid, IPC_RMID, 0) == -1) syserr("shmctl");
}   /* close_shared_mem_pipe */


read_shared_mem(fds, buf, size)
int fds, size;
char *buf;

{
    int *read_ptr, *write_ptr;
    char *rv, *wv, *start_addr, *max_addr;
    shared_mem_pipe *fd;

    fd = (shared_mem_pipe *) fds;

    P(fd->semid);
    read_ptr = (int *)(fd->addr);
    write_ptr = read_ptr +1;
    rv = (char *) *read_ptr;
    wv = (char *) *write_ptr;
    max_addr = (char *) (fd->addr + BUFSIZE - 1);
    start_addr = (char *) (read_ptr + 2);

    if (rv == max_addr) {
        if (rv != wv)
            { *buf = *max_addr; size = 1; *read_ptr = (int) start_addr; }
        else
            size = 0;
        }
    else {
        if (rv <= wv) size = min(wv - rv, size);
        else size = min(max_addr - rv, size);
        memcpy(buf, rv, size);
        *read_ptr = (int) (rv + size);
        }
```

```
    V(fd->semid);
    return(size);
}   /*  read_shared_memory  */


write_shared_mem(fds, buf, size)
int fds, size;
char *buf;

{
    int *read_ptr, *write_ptr;
    char *rv, *wv, *start_addr, *max_addr;
    shared_mem_pipe *fd;

    fd = (shared_mem_pipe *) fds;
    P(fd->semid);

    read_ptr = (int *)fd->addr;
    write_ptr = read_ptr +1;
    rv = (char *) *read_ptr;
    wv = (char *) *write_ptr;
    max_addr = (char *) (fd->addr + BUFSIZE - 1);
    start_addr = (char *) (read_ptr + 2);

    if (wv == max_addr) {
        if (rv == start_addr) size = 0;
        else
            { *wv = *buf; size = 1; *write_ptr = (int) start_addr;}
        }
    else {
        if (rv <= wv) size = min(max_addr - wv, size);
        else size = min(rv - wv - 1, size);
        memcpy(wv, buf, size);
        *write_ptr = (int) (wv + size);
        }
    V(fd->semid);
    return(size);
}   /*  write_shared_mem  */


/*************************  message  pipe  ********************/


create_message_pipe(fd)
int *fd;

{
    read_function = read_message;
    write_function = write_message;

    fd[0] = fd[1] = (int) fd;
```

```
    if((fd[0] = fd[1] = msgget(IPC_PRIVATE, 0666)) == -1)
        syserr("msgget");
    return ((int) fd);
}   /*  create_message_pipe  */


close_message_pipe(fds)
int fds;

{
    if(msgctl(fds, IPC_RMID, NULL) == -1)
        syserr("msgctl");
}   /*  close_message_pipe  */


read_message(fds, buf, size)
int fds, size;
char *buf;

{
 struct msgbuf *mbuf;

    if((mbuf = (struct msgbuf *) malloc(size + sizeof(mbuf->mtype))) == NULL)
        {
        syserr("malloc");
        return(0);
        };

    if((size = msgrcv(fds, mbuf, size, 0L, IPC_NOWAIT)) == -1)
        size = 0;

    memcpy(buf, ((char *) mbuf) + sizeof(mbuf->mtype), size);
    free(mbuf);
    return(size);
}   /*  read_message  */


write_message(fds, buf, size)
int fds, size;
char *buf;

{
    struct msgbuf *mbuf;

    if((mbuf = (struct msgbuf *) malloc(size + sizeof(mbuf->mtype))) == NULL) {
        syserr("malloc");
        return(0);
        }

    memcpy(((char *) mbuf) + sizeof(mbuf->mtype), buf, size);
    mbuf->mtype = 1;
```

```
    if(msgsnd(fds, mbuf, size, IPC_NOWAIT) == -1)
        size = 0;

    free(mbuf);
    return(size);
}   /* write_message */


/*************************** message pipe ********************/


create_pipe(fd)
int *fd;

{
    pipe(fd);
    read_function = read;
    write_function = write;
    return((int) fd);
}   /* create_pipe */


close_pipe(fds)
int fds;

{
    close(fds);
}   /* close_pipe */


/************************* general read/write ******************/

read_list(fds, head)
int fds;
node_struct **head;

{
int i, line_count = 0, line_length, a;
node_struct *cur, *new;
char *buf;

    *head  = cur = NULL;

    if ((a = safe_read(fds, &line_count, 4)) > 0) {
        for (i = 0; i < line_count; ) {
            if (safe_read(fds, &line_length, sizeof(int)) > 0) {
                buf = (char *)e_malloc(line_length);
                for ( ; safe_read(fds, buf, line_length) == 0; );
                new = (node_struct *)e_malloc(sizeof(node_struct));
                new->next = NULL;
                new->line = buf;
                if (*head == NULL)
```

```
                *head = new;
            else {
                for (cur = *head; cur->next != NULL; cur = cur->next);
                cur->next = new;
                }
            i++;
            }
        }   /* for i <= line_count  */
    }
    return(line_count);
}   /*  read_list  */


write_list(fds, head)
int fds;
node_struct *head;

{
int line_count, line_length;
node_struct *cur;

    line_count = 0;
    for (cur = head; cur != NULL; cur = cur->next)
        line_count++;
    safe_write(fds, &line_count, sizeof(int));
    for (cur = head; cur != NULL; cur = cur->next) {
        line_length = strlen(cur->line) + 1;
        safe_write(fds, &line_length, sizeof(int));
        safe_write(fds, cur->line, line_length);
        }
}   /*  write_list  */


read_list_delay(fds, head, delay)
int fds, delay;
node_struct **head;

/*  read a stream of character from fds file descriptor. if str is NULL
    then sleep 1 second and read again until time = delay */

{
int time;
node_struct *head1;

    head1 = NULL;
    for (time = 0; time < delay; time++) {
        if (read_list(fds, &head1) > 0)
            break;
        sleep(1);
        }
    *head = head1;
}   /*  read_list_delay  */
```

```
read_str(fds, str, ch)
int fds;
char **str, ch;

/*  head stream of character from fds file descriptor
    until NULL character is encountered */

{
int i=0, m_count=1, m_size;
char *temp;

    m_size = m_count * 100;
    *str = (char *)e_malloc(m_size);
    if (read(fds, *str, 1) != 0) {
        while (*(*str+i++) != ch) {
        /* if reach the end of buffer but not encountered
            new line yet then resize buffer */
            if (i == m_size) {
                temp = (char *)e_malloc(m_size);
                memcpy(*str, temp, i);
                m_size = ++m_count * 100;
                free(*str);
                *str = (char *)e_malloc(m_size);
                memcpy(*str, temp, i);
                free(temp);
                }       /* if i == m_size */
            read(fds, *str+i, 1);
            }   /* while */
        i--;
        *(*str+i) = '\0';
        }
    else
        **str = '\0';
    return(i);
}   /*  read_str  */


safe_read(fds, buf, nbytes)
int fds, nbytes;
char *buf;

{
int k, n=nbytes;
char *str;

    str = buf;
    if ((k = read_function(fds, buf, n)) > 0) {
        for (n-=k, buf+=k; n!=0; n-=k, buf+=k) {
            k = read_function(fds, buf, n);
            }
```

```
        return(nbytes);
        }
    else
        return(0);
}   /*  safe_read  */


safe_write(fds, buf, nbytes)
int fds, nbytes;
char *buf;

{
int k=0, n=nbytes;
char *str;

    str = buf;
    for ( ; n!=0; n-=k, buf+=k)
        k = write_function(fds, buf, n);
    return(nbytes);
}   /* safe_write  */
```

Appendix C

Control Library's Code

C.1 control.c

```
/*************************************************************/
/* Program      : control.h                                  */
/* Programmer   : Jeffrey Wijono                             */
/* description  : This file contains header file for control library */
/*************************************************************/

#define MAX_COLUMN      8
#define DELAY_TIME      16
#define False           0
#define True            1

typedef struct control_data {
        char  command[30];          /* command to be executed */
        char  key[40];              /* keyword matching      */
        int   heading;              /* boolean if the command will have a heading */
        int   delay;                /* duration time of teh command to be executed */
        int   col_n;                /* col of key to be matched  */
        int   col[MAX_COLUMN+1];    /* selected column to be displayed */
        int   fd_cntl_in[2];        /* file descriptor number for communication to */
        int   fd_cntl_out[2];       /* file descriptor number for communiation from */
        int   (*read_function)();   /* pointer to the read function  */
        int   (*write_function)();  /* pointer to the write function */
 } control_data_struct;

void initial_control_lib();
void set_process_command();
void set_delay_time();
void create_control_pipe();

extern int (*read_function)();
extern int (*write_function)();
```

C.2 control.c

```
/*************************************************************/
/*  program      : monitor.c                                 */
/*  programmer   :  Jeffrey Wijono                           */
/*  description  : Spawn the main process and set up the communication */
/*                   pipe for the main program and the monitor function */
/*************************************************************/

#include <stdio.h>

#include "miscellaneous.h"
```

```
#include "control.h"

control_data_struct *control_dat;
void initial_control_lib()

/* assign some default values for the control_dat global variable */

{
    control_dat = (control_data_struct *)malloc(sizeof(control_data_struct));
    strcpy(control_dat->command, "ps -ef");
    control_dat->delay = DELAY_TIME;
    control_dat->heading = False;
}   /*  initial_control_lib  */


void create_control_pipe()

/*  prepare the communication using the pipe file descriptor  */

{
    create_pipe(control_dat->fd_cntl_in);
    create_pipe(control_dat->fd_cntl_out);
    set_file_descriptor_no_delay(control_dat->fd_cntl_in);
    set_file_descriptor_no_delay(control_dat->fd_cntl_out);

    control_dat->read_function = read_function;
    control_dat->write_function = write_function;
} /*  create_control_pipe  */


void create_control_shared_mem_pipe()

/* prepare the communication of shared memory */

{

    create_shared_mem_pipe(control_dat->fd_cntl_in);
    sleep(1);
    create_shared_mem_pipe(control_dat->fd_cntl_out);
    sleep(1);
    control_dat->read_function = read_function;
    control_dat->write_function = write_function;
}   /* create_control_shared_mem_pipe */


void create_control_message_pipe()

/*  prepare the communication of message  */

{
    create_message_pipe(control_dat->fd_cntl_in);
    create_message_pipe(control_dat->fd_cntl_out);
```

```
    control_dat->read_function = read_function;
    control_dat->write_function = write_function;
} /* create_control_message_pipe */


void set_process_command(command)
char command[];

/* change the unix command to be processed */

{
    strcpy(control_dat->command, command);
} /* set_process_command */


void set_delay_time(delay_time)
int delay_time;

/* set the current duration time */

{
    control_dat->delay = delay_time;
} /* set_delay_time */


monitor(int *argc, char **argv)

/* monitor the process and use the message for the communication to the main process */
{
int f, c, pause;
node_struct *head;

    get_argv(argc, argv);
    create_control_pipe();

    if (f = fork(), f > 0) {  /* parent process */
        close_pipe(control_dat->fd_cntl_in[0]);
        close_pipe(control_dat->fd_cntl_out[1]);
        return(0);
        }
    else if (f == 0) {        /* child process */
        close_pipe(control_dat->fd_cntl_in[1]);
        close_pipe(control_dat->fd_cntl_out[0]);
        pause = False;  head = NULL;
        for (c = True; c == True; ) {
            c = process(head, &pause);
            clear_node(&head);
            if (control_dat->delay < 0) break;
            else
                read_list_delay(control_dat->fd_cntl_in[0], &head, control_dat->delay);
            }
        close_pipe(control_dat->fd_cntl_in[0]);
```

```
        close_pipe(control_dat->fd_cntl_out[1]);
        exit(0);
        }
    else
        syserr("First fork fail");
}   /* monitor  */


monitor_sm(int *argc, char **argv)

/*  monitor the process and use the message for the shared memory to the main process */

{
int f, c, pause;
node_struct *head;

    get_argv(argc, argv);
    create_control_shared_mem_pipe();

    if (f = fork(), f > 0) {   /* parent process */
        sleep(3);
        return(0);
        }
    else if (f == 0) {        /* child process */
        pause = False; head = NULL;
        for (c = True; c == True; ) {
            c = process(head, &pause);
            clear_node(&head);
            if (control_dat->delay < 0) break;
            else
                read_list_delay(control_dat->fd_cntl_in[0], &head, control_dat->delay);
            }
        close_shared_mem_pipe(control_dat->fd_cntl_in[0]);
        close_shared_mem_pipe(control_dat->fd_cntl_out[1]);
        exit(0);
        }
    else
        syserr("First fork fail");
}   /*  monitor_sm  */


monitor_message(int *argc, char **argv)

/*  monitor the process and use the message for the communication
    to the main process */
{
int f, c, pause;
node_struct *head;

    get_argv(argc, argv);
    create_control_message_pipe();
```

```
    if (f = fork(), f > 0) {   /* parent process */
        return(0);
        }
    else if (f == 0) {         /* child process */
        pause = False;  head = NULL;
        for (c = True; c == True; ) {
            c = process(head, &pause);
            clear_node(&head);
            if (control_dat->delay < 0) break;
            else
                read_list_delay(control_dat->fd_cntl_in[0], &head, control_dat->delay);
            }
        close_message_pipe(control_dat->fd_cntl_in[0]);
        close_message_pipe(control_dat->fd_cntl_out[1]);
        exit(0);
        }
    else
        syserr("First fork fail");
}   /*  monitor_message  */


process(head, pause)
node_struct *head;
int *pause;

/*  it will check what command send to the monitor process and check is it at
    pause mode. If it is not at pause mode then perform the unix command.  */

{
node_struct *cur;

    if (head) {
        if (strcmp(head->line, "_TERMINATE") == 0) {
            return(False);
            }
        else if (strcmp(head->line, "_CHANGE_COMMAND") == 0) {
            cur = head->next;
            strcpy(control_dat->command, cur->line);
            }
        else if (strcmp(head->line, "_CHANGE_DELAY") == 0) {
            cur = head->next;
            control_dat->delay = atoi(cur->line);
            }
        else if (strcmp(head->line,"_PAUSE") == 0) {
            *pause = True;
            }
        else if (strcmp(head->line,"_RESUME") == 0) {
            *pause = False;
            }
        }
     if (*pause == False) {
        monitor_system(control_dat->command);
```

```
        }
    return(True);
}   /*   process  */


monitor_system(command)
char *command;

/*  This function simulated the regular system command. It sends the output of this
    command into a file descriptor and the output is already in the linked list format. */

{
node_struct *head, *cur, *new;
int fd[2], f;
char *str;

    pipe(fd);

    if (f = fork(), f == 0) {     /* child process   */
        close_pipe(fd[0]);
        close_pipe(STD_OUTPUT);
        dup(fd[1]);
        system(command);
        exit(0);
        }
    else if (f > 0) {          /* parent process  */
        close_pipe(fd[1]);
        head = NULL;
        for ( ; read_str(fd[0], &str, '\n') > 0; ) {
            new = (node_struct *)malloc(sizeof(node_struct));
            new->next = NULL;
            new->line = str;
            if (head == NULL)
                head = new;
            else {
                for (cur = head; cur->next != NULL; cur = cur->next);
                cur->next = new;
                }
            }
        write_list(control_dat->fd_cntl_out[1], head);
        clear_node(&head);
        close_pipe(fd[0]);
        }           /* parent process */
    else
        syserr("monitor_system fork fail");
}   /* monitor_system */


receive_from_control(head)
node_struct **head;

/*  receive a linked list of data from the control process  */
```

```
{
    node_struct *head1;
    int n;

    read_function = control_dat->read_function;
    *head = head1 = NULL;
    n = read_list(control_dat->fd_cntl_out[0], &head1);
    *head = head1;
    return (n);
}   /* receive_from_control */


send_to_control(head)
node_struct *head;

/*  send a linked list of data to the control process  */

{
    write_function = control_dat->write_function;
    write_list(control_dat->fd_cntl_in[1], head);
}   /* send_to_control */


get_argv(int *argc, char *argv[])

/*  parse the argv and remove the selected command from the argv  */

{
int i, n, delay;

    for (n = 1; n < *argc; n++) {
        if (strcmp(argv[n], "_delay") == 0) {
        delay = atoi(argv[n+1]);
        set_delay_time(delay);
        (*argc)-=2;
        for (i=n; i < (*argc); i++)
            argv[i] = argv[i+2];
        n = 1;
        }
    if (strcmp(argv[n], "_command") == 0) {
        set_process_command(argv[n+1]);
        (*argc)-=2;
        for (i=n; i < (*argc); i++)
            argv[i] = argv[i+2];
        n = 0;
        }
    }   /*  for n < *argc */
}   /*  get_argv */
```

Appendix D

Transform Library's Code

D.1 trans.h

```
/*************************************************************/
/* Program      : trans.h                                    */
/* Programmer   : Jeffrey Wijono                             */
/* description  : This file contains header file for transform library */
/*************************************************************/

typedef struct loc {
        char        name[50];           /* title of the column */
        int         min, max;           /* min and max position of each column */
        } loc_struct;

typedef struct column_data {
        char                *content;   /* the content of each column */
        struct column_data  *next_col;  /* pointer to the next column */
        } column_data_struct;

typedef struct row_col {
        column_data_struct  *col_head;  /* column pointer */
        struct row_col      *next_row;  /* pointer to the next row */
        } row_col_struct;

typedef struct trans_data {
        int     fd_trans[2];            /* file desriptor */
        int     col[MAX_COLUMN+1];      /* column to be displayed */
        void    (*func)();              /* function to be invoked when menu is selected */
        row_col_struct  *row_col_head;  /* pointer to the head of data */
        } trans_data_struct;


typedef struct _pid_list {
        int_node        *list;          /* pointer to the list of pid_struct */
        struct _pid_list  *next;        /* pointer to the next node in the list */
        } pid_list_struct;


trans_data_struct   *initial_trans_lib();
row_col_struct      *trans_to_row_col();
node_struct         *create_display_list();
void                clear_row_col_list();
```

D.2 transform.c

```
/*************************************************************/
/* program      : transform.c                                */
/* programmer   : Jeffrey Wijono                             */
/* description  : This file contains functions for transformation */
/*************************************************************/


#include <stdio.h>
#include <string.h>
#include "miscellaneous.h"
#include "trans.h"

row_col_struct parse();
trans_data_struct *trans_dat;


trans_data_struct *initial_trans_lib()

/* initial the global variable trans_dat to some default values  */

{
int fd_trans[2], i;

    pipe(fd_trans);
    trans_dat->fd_trans[0] = fd_trans[0];
    trans_dat->fd_trans[1] = fd_trans[1];
    for (i=0; i<MAX_COLUMN; i++)
        trans_dat->col[i] = i + 1;
    trans_dat->col[MAX_COLUMN] = 0;
    return(trans_dat);
}   /*  initial_trans_lib   */


void set_trans_function(func)
void (*func)();

/* set the transform function to the right trans function  */

{
    trans_dat->func = func;
}   /* set_trans_function  */


row_col_struct *trans_to_row_col(head)
node_struct *head;

/* transform a list of rows into a list of rows columns */

{
    node_struct *cur;
```

```
    row_col_struct *row_col_head, *cur_row_col, *new_row_col;
    column_data_struct *new_col, *cur_col;
    loc_struct location[MAX_COLUMN+1];
    char *temp, *row;
    int num_of_col, i, str_len, lines = 0;
    get_location(head, location, &num_of_col);
    row_col_head = NULL;
    for (cur = head; cur != NULL; cur = cur->next) {
        new_row_col = (row_col_struct *)e_malloc(sizeof(row_col_struct));
        new_row_col->col_head = NULL;
        new_row_col->next_row = NULL;
        row = cur->line;
        lines++;
        for (i = 0; i < num_of_col; i++) {
            new_col = (column_data_struct *)e_malloc(sizeof(column_data_struct));
            new_col->next_col = NULL;
            temp =  row+location[i].min;
            if (i == num_of_col - 1) {  /* if the last column  */
                str_len = strlen(temp);
                new_col->content = (char *)e_malloc(str_len+1);
                strcpy(new_col->content, temp);
                }
            else {
                str_len = location[i].max - location[i].min + 1;
                new_col->content = (char *)e_malloc(str_len+1);
                strncpy(new_col->content, temp, str_len);
                }
            new_col->content[str_len] = '\0';
            if (new_row_col->col_head == NULL)
                new_row_col->col_head = new_col;
            else {
                for (cur_col = new_row_col->col_head; cur_col->next_col != NULL;
                    cur_col = cur_col->next_col);
                    cur_col->next_col = new_col;
                }
            }   /* for i < num_of_col */
        if (row_col_head == NULL)
            row_col_head = new_row_col;
    else {
        for (cur_row_col=row_col_head; cur_row_col->next_row != NULL;
            cur_row_col = cur_row_col->next_row);
            cur_row_col->next_row = new_row_col;
        }   /* else          */
    }   /* end of for loop  */
    return(row_col_head);
}       /* trans_to_row_col */


get_location(head, location, num_of_col)
node_struct *head;
loc_struct location[MAX_COLUMN+1];
int *num_of_col;
```

```
/* figure out the location of the column  */

{
node_struct *cur;

    get_first_loc(head->line, location, num_of_col);
    for (cur = head->next; cur != NULL; cur = cur->next)
        get_max_loc(cur->line, location, num_of_col);
}   /* get_location   */


void clear_row_col_list(row_col_head)
row_col_struct **row_col_head;

/*  empty the row_col_list and release the memory use by the list  */

{
row_col_struct *cur_row, *prev_row;
column_data_struct *cur_col, *prev_col;

    prev_row = NULL;
    for (cur_row = *row_col_head; cur_row != NULL; ) {
        prev_col = NULL;
        for (cur_col = cur_row->col_head; cur_col != NULL; ) {
            prev_col = cur_col;
            cur_col = cur_col->next_col;
            free(prev_col->content);
            free(prev_col);
            }
        prev_row = cur_row;
        cur_row = cur_row->next_row;
        free(prev_row);
        }   /* for cur_row != NULL */
    *row_col_head = NULL;
}   /* clear_row_col_list  */


node_struct *create_display_list(row_col_head, key, cols, col_of_key, pid_list_head,
select_head)
row_col_struct *row_col_head;
char *key;
int cols[MAX_COLUMN+1], col_of_key;
pid_list_struct **pid_list_head;
int_node *select_head;

/*  create a new list that contains a string for each node, this string
    means a row output. */

{
char *buf, *temp, *str2;
row_col_struct *cur_row;
```

```
column_data_struct *cur_col;
node_struct *head, *new;
int n, m_size, m_count, old_size, c, insert_cond, first_row, pid;
int_node *pid_head, *cur_pid;
pid_list_struct *new_list, *cur;

    head = NULL;
    pid_head = NULL;
    first_row = TRUE;
    for (cur_row = row_col_head; cur_row != NULL;
        cur_row = cur_row->next_row) {
        if (col_of_key > 0) {  /* if there is a keyword matching */
            if (first_row)
                insert_cond = TRUE;
            else {
                insert_cond = FALSE;
                for (c=1, cur_col=cur_row->col_head; cur_col!=NULL;
                    cur_col=cur_col->next_col, c++) {
                    str2 = (char *)e_malloc(strlen(cur_col->content)+1);
                    skipblanks(cur_col->content, str2);
                    if (c==col_of_key && strcmp(key, str2) == 0) {
                        insert_cond = TRUE;
                        break;
                        }
                    free(str2);
                    }
                }  /* if not the first row     */
            }  /* if there is a select key */
        else
            insert_cond = TRUE;
            /* select the desired column only */
        if (insert_cond) {
            new = (node_struct *)e_malloc(sizeof(node_struct));
            new->next = NULL;
            m_count = 1;
            m_size = m_count * COL_WIDTH;
            buf = (char *)e_malloc(m_size);
            *buf = ' ';
            *(buf+1) = '\0';
            n = 0;
            for (c = 1, cur_col = cur_row->col_head; cur_col != NULL;
                cur_col = cur_col->next_col, c++) {
                if (c == 2) {  /* pid column */
                    pid = atoi(cur_col->content);
                    }
                if (cols[n] == c) {
                    if (strlen(cur_col->content)+strlen(buf) > m_size) {
                        temp = (char *)e_malloc(m_size);
                        memcpy(temp, buf, m_size);
                        free(buf);
                        old_size = m_size;
                        m_size = ++m_count * COL_WIDTH;
```

```
                        buf = (char *)e_malloc(m_size);
                        memcpy(buf, temp, old_size);
                        free(temp);
                        } /* if to create a bigger buffer */
                    strcat(buf, cur_col->content);
                    strcat(buf, "  ");
                    n++;
                    }   /* cols[n] == c */
                }   /* for cur_col != NULL */
            new->line = buf;
            if (select_head != NULL) {
                if (first_row)
                    insert_node(&head, new);
                else
                    if (match_pid(select_head, pid)) {
                        insert_node(&head, new);
                        insert_int_node(&pid_head, pid);
                        }
                }
            else {  /*  if select_head == NULL */
                insert_node(&head, new);
                insert_int_node(&pid_head, pid);
                }
            }  /*   if insert cond   */
        first_row = FALSE;
        }   /* for cur != NULL     */
    new_list = (pid_list_struct *)e_malloc(sizeof(pid_list_struct));
    new_list->list = pid_head;
    new_list->next = NULL;
    if (*pid_list_head == NULL)
        *pid_list_head = new_list;
    else {
        for (cur = *pid_list_head; cur->next != NULL; cur = cur->next);
        cur->next = new_list;
        }
    return (head);
}   /* create_display_list */


match_pid(head, pid)
int_node *head;
int pid;

/*  Look for pid in the list of pid  */

{
int_node *cur;

    for (cur = head; cur != NULL; cur = cur->next)
        if (pid == cur->num)
            return(TRUE);
    return(FALSE);
```

```
}   /*  match_pid  */


insert_pid_str(head, str)
int_node **head;
char *str;

/* insert str at the end of the list pointed by head  */

{
int_node *new, *cur;

    new = (int_node *)e_malloc(sizeof(int_node));
    new->num = atoi(str);
    new->next = NULL;
    if (*head == NULL)
        *head = new;
    else {
        for (cur = *head; cur->next != NULL; cur = cur->next);
        cur->next = new;
    }
}   /*  insert_pid_str  */


insert_pid_list(head, alist)
pid_list_struct **head;
int_node *alist;

/* insert alist at the end of list pointed by head  */

{
pid_list_struct *new, *cur;

    new = (pid_list_struct *)e_malloc(sizeof(pid_list_struct));
    new->list = alist;
    new->next = NULL;
    if (*head == NULL)
        *head = new;
    else {
        for (cur = *head; cur->next != NULL; cur = cur->next);
        cur->next = new;
        }
}   /*  insert_pid_list  */


str_to_array(str, cols)
char *str;
int cols[MAX_COLUMN+1];

/* convert a string into an array of integer */

{
```

```
int i;
loc_struct location[MAX_COLUMN+1];

    for (i=0; i<=MAX_COLUMN; i++)
        location[i].name[0] = '\0';
    parse_string(str, location);
    for (i=0; i<=MAX_COLUMN; i++) {
        if (strlen(location[i].name)>0)
            cols[i] = atoi(location[i].name);
        else
            cols[i] = 0;
        }
}   /* str_to_array */


skipblanks(str1, str2)
char *str1, *str2;

/* clear all the white space in the beginning of the string */

{
char *p;

    p = strtok(str1, " ");
    strcpy(str2, p);
}   /* skipblanks  */


get_first_loc(str, local, num)
char *str;
loc_struct local[MAX_COLUMN+1];
int *num;

/* get the begining and ending position of each collumn heading */

{
int i, n;

    for (n=0; n<=MAX_COLUMN; n++)
        local[n].min = local[n].max = 0;
    i = 0;
    for (n = 0; str[n] != '\0'; n++) {
        if (str[n] != ' ') {
            local[i].min = n;
            for (; *(str+n+1) != ' ' && *(str+n+1) != '\0'; n++);
            local[i].max = n;
            i++;
            }   /*  if   */
        }   /*  for  */
    *num = i;
    local[i].min = local[i].max = COL_WIDTH;
}   /* get_first_loc */
```

```
int distance(start, last, loc, i)
int start, last, i;
loc_struct loc[MAX_COLUMN+1];

/*  figure out the distance of two string
    The distance is negative if one of the string is over lap the other */

{
    if (last <= loc[i].max)
        return(loc[i].min  - last);
    else
        return(start - loc[i].max);
}   /* distance */


parse_string(str, location)
char *str;
loc_struct location[MAX_COLUMN+1];

/* break a string into array of string using white space as a deliminator */

{
char *p, *temp;
int n;

    for (n=0; n<=MAX_COLUMN; n++)
        location[n].name[0] = '\0';
    n=0;
    temp = (char *)malloc(strlen(str)+1);
    strcpy(temp, str);
    p = strtok(temp, " ");
    do {
        strcpy(location[n].name, p);
        p = strtok('\0', ", ");
        n++;
        } while (p);
}       /* parse_string   */


get_max_loc(str, loc, num)
char *str;
loc_struct loc[MAX_COLUMN+1];
int *num;

/*  get the begining and ending position of each collumn in the table by
    go through the whole table and using distance formula. */

{
int i, n, start, last, dist, dist2;
```

```
    start = last = i = 0;
    for (n = 0; str[n] != '\0'; n++) {
        if (str[n] != ' ') {
            if (i == *num) {
                if (n < loc[i].min)
                loc[i].min = n;
                for (; str[n+1] != '\0'; n++);
                loc[i].max = n;
                }
            else {
                start = n;
                for (; str[n+1] != ' ' && str[n+1] != '\0'; n++);
                last = n;
                dist = distance(start, last, loc, i); /* figure out the right column */
                if (dist > 0) {
                    dist2 = distance(start, last, loc, i+1);
                    if (dist2 > 0) {       /* new string in between of two column */
                        if (dist > dist2) {  /* if closer to the next column */
                            i++;
                            loc[i].min = start;
                            }
                        else               /* if closer to the current column */
                            loc[i].max = last;
                        }
                    else {                 /* overlap with next column */
                        i++;
                        if (loc[i].min > start)
                            loc[i].min = start;
                        if (loc[i].max < last)
                            loc[i].max = last;
                        }
                    }  /* if  */
                else {
                    if (loc[i].min > start)
                        loc[i].min = start;
                    if (loc[i].max < last) {
                        loc[i].max = last;
                        if (loc[i+1].min <= last) {
                            loc[i+1].max = last + loc[i+1].max - loc[i+1].min + 2;
                            loc[i+1].min = last + 2;
                            }
                        }
                    }  /* if dist <= 0  */
                }  /* if i != *num  */
            }  /* if str[n] != ' '  */
        else {  /* if str[n] == ' '  */
            if (i != 0 && n >= loc[i+1].min && i < *num)
                i++;
            } /* else */
        }  /* for */
}  /* get_max_loc */
```

Appendix E

Display Library's Code

E.1 display.h

```
/*************************************************************/
/* Program      : display.h                                  */
/* Programmer   : Jeffrey Wijono                             */
/* description  : This file contains header file for display library */
/*************************************************************/

#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>

#define SNAP_SIZE   3200
#define FIRST          1
#define LAST           0

typedef struct _menu_data {
        char        *name;              /* name on menubar   */
        void        (*func)();          /* callback function */
        int         n_sub_items;        /* number of items at the submenu */
        char        *sub_menu_title;    /* title of sub_menu  */
        struct _menu_data *next;        /* pointer to the menu in the menubar */
        structmenu_data *sub_menu;
        } menu_struct;

typedef struct _menu_list {
        menu_struct       *menu;        /*  name of the current menu */
        struct _menu_list  *next;       /*  pointer to the next menu list on the list */
        } menu_list_struct;

typedef struct _display_data {
        Widget      toplevel;           /* toplevel Widget  */
        Widget      aForm;              /* after the toplevel */
        Widget      menubar;            /* menubar widget   */
        int         fd_disp_in[2];      /* file descriptor from main */
        int         fd_disp_out[2];     /* file descriptor out to main */
        int         (*read_function)();     /* pointer to the reading function */
        int         (*write_function)();    /* pointer to the writing function */
        int         (*close_communication)(); /* pointer to close communication */
        XtWorkProcId work_proc_id;
        } display_win_struct;

typedef struct _display_list {
        Widget      title;              /* title display */
        Widget      display;            /* scrolled list  */
        Widget      vscroll;            /* vertical scrolled bar */
        Widget      hscroll;            /* horizontal scrolled bar */
```

```
        int         pause;              /* pause condition */
        } display_list_struct;

typedef struct _display_text {
        Widget      title;              /* title display */
        Widget      display;            /* scrolled list */
        Widget      vscroll;            /* vertical scrolled bar */
        Widget      hscroll;            /* horizontal scrolled bar */
        int         pause;              /* pause condition */
        } display_text_struct;

typedef struct _graph_data {
        Widget      title;              /* title display */
        Widget      canvas;             /* graphics display window */
        char        title_name[100];    /* title name of the current display */
        int         depth;              /* number of screen depth */
        int         ncolors;            /* number of colors on the display */
        int         granulation;        /* max number of bar on the graph window */
        int         count;              /* number of data */
        int         scale;              /* the maximum number of the bar graph */
        int         foreground;         /* foreground color */
        int         background;         /* background color */
        int_node    *head;              /* pointer to data list */
        GC          gc;                 /* graphics context */
        Dimension   width;              /* width of the graph window */
        int         height              /* height of the graph window */
        Pixmap      pix;                /* buffer for the current display */
        } display_graph_struct;

typedef struct _dialog_data{
        Widget      label;              /* parent widget of this dialog  */
        Widget      value[3];           /* widget for the value of this dialog */
        int         num_of_col;         /* the current number of columns */
        short int   type;               /*  type of display widget  (text/list) */
        } dialog_data_struct;

void        initial_list_display();
void        create_display_pipe();
void        disp_main_loop();
void        initialize_menu();
void        menu();
void        menu_item();
void        done_menu();
void        display_title();
void        display_list_data();
void        display_graph_data();
void        refresh();
void        resize();
void        ClearList();
void        AddToList();
void        draw_line();
void        draw_data();
```

```
void        draw_filled_rectangle();

Widget      get_toplevel();
Widget      get_menubar();
Widget      get_display();
Widget      CreateFormManagedWidget();
Widget      CreateLabel();
Widget      CreateMenubar();
Widget      CreateScrolledList();
Widget      CreateScrolledText();
Widget      CreateSingleLineText();
Widget      DispGetScrolledWidget();

void        DoSaveList();
void        DoSaveText();
void        DoPrintList();
void        DoPrintText();
void        DoQuit();
void        DoQuit2();
void        DoUndo();
void        DoCommand();
void        DoDelay();
void        DoPause();
void        DoResume();
void        DoKillProcess();
void        DoKeySelect();
void        DoColSelect();
void        DoRowSelect();
void        UndoRowSelect();
void        DoSnap();
void        DoGranulation();
void        DoSetTitle();
void        DoScale();
void        text_snap();

extern int  (*read_function)();
extern int  (*write_function)();
```

E.2 display.c

```
/*************************************************************/
/* program      : display.c                                  */
/* programmer   : Jeffrey Wijono                             */
/* description  : display library                            */
/*                pipe for the main program and the monitor function */
/*************************************************************/

#include <stdio.h>

#include "miscellaneous.h"
#include "display.h"
```

```
#include "communication.h"

#include <X11/Shell.h>
#include <X11/cursorfont.h>
#include <X11/Xutil.h>
#include <Xm/Xm.h>
#include <Xm/BulletinB.h>
#include <Xm/CascadeB.h>
#include <Xm/DrawingA.h>
#include <Xm/Form.h>
#include <Xm/Label.h>
#include <Xm/List.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
#include <Xm/Separator.h>
#include <Xm/Text.h>
#include <Xm/ToggleB.h>

/*
 * global variables in the display library
 */

display_win_struct     *display_dat;
display_list_struct    *display_list_dat;
display_text_struct    *display_text_dat;
display_graph_struct   *display_graph_dat;

/*
 * Functions
 */

void initial_list_display()

{
    display_dat = (display_win_struct *)e_malloc(sizeof(display_win_struct));
    display_list_dat = (display_list_struct *)e_malloc(sizeof(display_list_struct));
    display_list_dat->title = NULL;
    display_list_dat->pause = False;
}   /* initial_list_display */


void initial_text_display()

{
    display_dat = (display_win_struct *)e_malloc(sizeof(display_win_struct));
    display_text_dat = (display_text_struct *)e_malloc(sizeof(display_text_struct));
    display_text_dat->title = NULL;
}   /* initial_text_display */


void initial_graph_display()
```

```
{
    display_dat = (display_win_struct *)e_malloc(sizeof(display_win_struct));
    display_graph_dat = (display_graph_struct *)e_malloc(sizeof(display_graph_struct));
    display_graph_dat->title = NULL;
    display_graph_dat->scale = 100;
    display_graph_dat->granulation = 10;
    display_graph_dat->count = 0;
}   /* initial_graph display  */


void create_display_pipe()

{
    create_pipe(display_dat->fd_disp_in);
    create_pipe(display_dat->fd_disp_out);
    set_file_descriptor_no_delay(display_dat->fd_disp_in);
    set_file_descriptor_no_delay(display_dat->fd_disp_out);
    display_dat->read_function = read_function;
    display_dat->write_function = write_function;
    display_dat->close_communication = close_pipe;
}


void create_display_shared_mem_pipe()

{
    create_shared_mem_pipe(display_dat->fd_disp_in);
    create_shared_mem_pipe(display_dat->fd_disp_out);
    display_dat->read_function = read_function;
    display_dat->write_function = write_function;
    display_dat->close_communication = close_shared_mem_pipe;
}   /* create_display_shared_mem_pipe */


void create_display_message_pipe()

{
    create_message_pipe(display_dat->fd_disp_in);
    create_message_pipe(display_dat->fd_disp_out);
    display_dat->read_function = read_function;
    display_dat->write_function = write_function;
    display_dat->close_communication = close_message_pipe;
}   /* create_display_message_pipe */


void set_title(title_string)
char *title_string;

{
    strcpy(display_graph_dat->title_name, title_string);
}   /* set_title  */
```

```
void set_graph_scale(data)
int data;

{
    display_graph_dat->scale = data;
}   /* set_graph_scale  */


void set_graph_granulation(data)
int data;

{
    display_graph_dat->granulation = data;
}   /* set_graph_granulation  */


void create_list_display(argc, argv)
int argc;
char *argv[];

/*  create an output window which contains a list widget and use pipe
    for the communication  */

{
int f;
XtWorkProcId work_proc_id = NULL;

create_display_pipe();
if (f = fork(), f == 0) {    /* child process */
/* close(display_dat->fd_disp_in[1]);
    close(display_dat->fd_disp_out[0]); */
    display_dat->toplevel = XtInitialize(argv[0], "Jeff",
                                NULL, 0, &argc, argv);
    display_dat->aForm = CreateFormManagedWidget(display_dat->toplevel,
                                "aForm", 510, 390);
    display_dat->menubar = CreateMenubar(display_list_dat, "Menubar");
    display_list_dat->title   = CreateLabel(display_dat->aForm, "Title", 40);
    display_list_dat->display = CreateScrolledList(display_dat->aForm, "Display", 80, 17);
    display_list_dat->vscroll = DispGetScrolledWidget(display_list_dat->display);

    XtManageChild(display_dat->menubar);
    XtManageChild(display_list_dat->title);
    XtManageChild(display_list_dat->display);

    display_dat->work_proc_id = XtAddWorkProc(display_list_data, NULL);

    XtRealizeWidget(display_dat->toplevel);
    XtMainLoop();
    }
else if (f > 0) {           /* parent process */
    return;
```

```
    }
else
    syserr("Fail on display fork()\n");
}   /* create_list_display */


void create_list_display_message(argc, argv)
int argc;
char *argv[];

/*  create an output window which contains a list widget and use message for
    the communication  */

{
int f;
XtWorkProcId work_proc_id = NULL;

create_display_message_pipe();
if (f = fork(), f == 0) {    /* child process */

    display_dat->toplevel = XtInitialize(argv[0], "Jeff",
                                         NULL, 0, &argc, argv);
    display_dat->aForm = CreateFormManagedWidget(display_dat->toplevel,
                                         "aForm", 510, 390);
    display_dat->menubar = CreateMenubar(display_list_dat, "Menubar");
    display_list_dat->title  = CreateLabel(display_dat->aForm, "Title", 40);
    display_list_dat->display = CreateScrolledList(display_dat->aForm, "Display", 80, 17);
    display_list_dat->vscroll = DispGetScrolledWidget(display_list_dat->display);

    XtManageChild(display_dat->menubar);
    XtManageChild(display_list_dat->title);
    XtManageChild(display_list_dat->display);

    display_dat->work_proc_id = XtAddWorkProc(display_list_data, NULL);

    XtRealizeWidget(display_dat->toplevel);
    XtMainLoop();
    }
else if (f > 0) {           /* parent process */
    return;
    }
else
    syserr("Fail on display fork()\n");
}   /* create_list_display  */


void create_text_display(argc, argv)
int argc;
char *argv[];

/*  create an output display which contains a text widget  */
```

```
{

    display_dat->toplevel = XtInitialize(argv[0], "Jeff",
                                         NULL, 0, &argc, argv);
    display_dat->aForm = CreateFormManagedWidget(display_dat->toplevel,
                                         "aForm", 510, 390);
    display_dat->menubar = CreateMenubar(display_dat->aForm, "Menubar");
    display_text_dat->title  = CreateSingleLineText(display_dat->aForm, "Title", 40);
    display_text_dat->display = CreateScrolledText(display_dat->aForm, "Display", 80,
                                17);
    display_text_dat->vscroll = DispGetScrolledWidget(display_text_dat->display);

    XtManageChild(display_dat->menubar);
    XtManageChild(display_text_dat->title);
    XtManageChild(display_text_dat->display);

    text_snap(argc,argv);
    XtRealizeWidget(display_dat->toplevel);
    XtMainLoop();
}   /*create_text_display */


Widget DispGetScrolledWidget(w)
Widget w;

{
Arg wargs[1];
Widget vscroll;

    XtSetArg(wargs[0],XmNverticalScrollBar, &vscroll);
    XtGetValues(w, wargs, 1);
    return vscroll;
}


void display_title(title)
char *title;

{
XmString xmstr;
Arg   wargs[1];

    xmstr = XmStringCreateLtoR(title, XmSTRING_DEFAULT_CHARSET);
    XtSetArg(wargs[0], XmNlabelString, xmstr);
    XtSetValues(display_list_dat->title, wargs, 1);
    XtFree(xmstr);
}


void display_list_data(Widget w)

/*  create the output window which contains list widget  */
```

```
{
char *buf;
int select_count, value, line_count = 0, length, received = 0, n;
Arg wargs[1];
node_struct *head, *head1, *cur, *title;

    head = NULL;
    if ((line_count = read_list(display_dat->fd_disp_in[0], &head)) > 0) {
    XtRemoveWorkProc(display_dat->work_proc_id);
    display_dat->work_proc_id = XtAddWorkProc(display_list_data, NULL);

    head1 = NULL;
    if (strcmp(head->line, "_DATA_") == 0) {
        XtSetArg(wargs[0], XmNvalue, &value);
        XtGetValues(display_list_dat->vscroll, wargs, 1);
        XtSetArg(wargs[0], XmNselectedItemCount, &select_count);
        XtGetValues(display_list_dat->display, wargs, 1);
        if (select_count > 0 || value > 0) {
            head1 = (node_struct *)e_malloc(sizeof(node_struct));
            head1->line = (char *)e_malloc(20);
            head1->next = NULL;
            strcpy(head1->line, "_PAUSE");
            write_list(display_dat->fd_disp_out[1], head1);
            clear_node(&head1);
            received++;
            }
        else if (display_list_dat->pause) {
            received++;    /* received data but not display */
            }
    else {
        for (n=0; n<=received; n++) {
            head1 = (node_struct *)e_malloc(sizeof(node_struct));
            head1->line = (char *)malloc(20);
            head1->next = NULL;
            strcpy(head1->line, "_CLEAR_LIST_DISPLAY");
            write_list(display_dat->fd_disp_out[1], head1);
            clear_node(&head1);
            }
        ClearList(display_list_dat->display);
        title = head->next;
        display_title(title->line);
        for (cur = title->next; cur != NULL; cur = cur->next) {
            AddToList(display_list_dat->display, cur->line, LAST);
            }
        }
    } /* if data is send to display */
    else if (strcmp(head->line, "_COMMAND_") == 0) {
        cur = head->next;
        if (is_command(cur, "_TERMINATE") == 0) {
            XtCloseDisplay(XtDisplay(display_dat->aForm));
            display_dat->close_communication(display_dat->fd_disp_in[0]);
```

```
            display_dat->close_communication(display_dat->fd_disp_out[1]);
            exit(0);
            }
        }
    else if (strcmp(head->line, "_FUNCTION") == 0) {
        for (; safe_read(display_dat->fd_disp_in[0], &length, sizeof(int)) == 0; );
        buf = (char *)e_malloc(length);
        for (; safe_read(display_dat->fd_disp_in[0], buf, length) == 0; );
        }
    clear_node(&head);
    }   /* if line_count > 0 */
}       /* display_list_data  */


void create_graph_display(argc, argv)
int argc;
char *argv[];

/*  create a graph widget  */

{
int f, n;
Arg wargs[6];

create_display_pipe();
if (f = fork(), f == 0) {    /* child process */
    display_dat->toplevel = XtInitialize(argv[0], "Jeff",
                                         NULL, 0, &argc, argv);
    display_dat->aForm = CreateFormManagedWidget(display_dat->toplevel,
                                         "aForm", 300, 260);

    display_dat->menubar = CreateMenubar(display_list_dat, "Menubar");
    if (display_graph_dat->title_name) {
        n = 0;
        XtSetArg(wargs[n], XmNtopAttachment,  XmATTACH_FORM);  n++;
        XtSetArg(wargs[n], XmNleftAttachment,  XmATTACH_WIDGET);  n++;
        XtSetArg(wargs[n], XmNleftWidget, display_dat->menubar);  n++;
        XtSetArg(wargs[n], XmNrightAttachment,  XmATTACH_FORM); n++;

        display_graph_dat->title =
            XtCreateManagedWidget ("Title",
                xmLabelWidgetClass, display_dat->aForm, wargs, n);
        XtManageChild(display_graph_dat->title);
        }

    n = 0;
    XtSetArg(wargs[n], XmNtopAttachment,  XmATTACH_WIDGET);  n++;
    XtSetArg(wargs[n], XmNtopWidget,  display_dat->menubar);  n++;
    XtSetArg(wargs[n], XmNbottomAttachment,XmATTACH_FORM);  n++;
    XtSetArg(wargs[n], XmNleftAttachment,  XmATTACH_FORM);n++;
    XtSetArg(wargs[n], XmNrightAttachment,  XmATTACH_FORM); n++;
```

```
    display_graph_dat->canvas = XtCreateManagedWidget("canvas",
                                        xmDrawingAreaWidgetClass,
                                        display_dat->aForm, wargs, n);
    init_canvas_data(display_graph_dat->canvas);

    XtManageChild(display_dat->menubar);
    XtRealizeWidget(display_dat->toplevel);

    XtAddCallback(display_graph_dat->canvas, XmNexposeCallback, refresh, NULL);
    XtAddCallback(display_graph_dat->canvas, XmNresizeCallback, resize, NULL);
    display_dat->work_proc_id = XtAddWorkProc(display_graph_data, NULL);

    XtMainLoop();
    }
else if (f > 0) {  /* parent process */
    return;
    }
else
    syserr("Fail on display fork()\n");
}   /*    create_graph_display    */


void display_graph_data(w)
Widget w;

{
char *buf;
int data, length, line_count;
node_struct *head, *cur;

    head = NULL;
    if ((line_count = read_list(display_dat->fd_disp_in[0], &head)) > 0) {
        XtRemoveWorkProc(display_dat->work_proc_id);
        display_dat->work_proc_id = XtAddWorkProc(display_graph_data, NULL);
        if (strcmp(head->line, "_DATA_") == 0) {
            cur = head->next;
            data = atoi(cur->line);
            draw_data(display_graph_dat->canvas, data);
            }   /* if data is send to display */
        else if (strcmp(head->line, "_COMMAND_") == 0) {
            cur = head->next;
            if (is_command(cur, "_TERMINATE") == 0) {
                XtCloseDisplay(XtDisplay(display_dat->aForm));
                exit(0);
                }
            }   /* else if _COMMAND_ */
        clear_node(&head);
        }   /* if line_count > 0   */
}   /* display_graph_data   */


init_canvas_data(w)
```

```
Widget        w;

{
XGCValues values;
Arg      wargs[5];
/*
* Get the colors the user has set for the widget.
*/
    XtSetArg(wargs[0], XtNforeground, &display_graph_dat->foreground);
    XtSetArg(wargs[1], XtNbackground, &display_graph_dat->background);
    XtSetArg(wargs[2], XtNwidth, &display_graph_dat->width);
    XtSetArg(wargs[3], XtNheight, &display_graph_dat->height);
    XtGetValues(w, wargs,4);
    /*
    * Fill in the values structure
    */
    values.foreground = display_graph_dat->foreground;
    values.background = display_graph_dat->background;
    values.fill_style = FillTiled;
    /*
    * Get the GC used for drawing.
    */
    display_graph_dat->gc= XtGetGC(w, GCForeground | GCBackground |
                                        GCFillStyle, &values);
    XSetForeground(XtDisplay(w), display_graph_dat->gc,
                                        BlackPixelOfScreen(XtScreen(w)));
}   /* init_canvas_data  */


void draw_data(w, data)
Widget  w;
int     data;

{
int i, bar_value, bar_width, height_factor, x_pos;
int_node *cur;

    insert_int_node(&(display_graph_dat->head), data);

    bar_width = (display_graph_dat->width -
        (display_graph_dat->granulation + 1)) / display_graph_dat->granulation;
    x_pos = (display_graph_dat->count - 1) * (bar_width + 1) + 2;

    if (display_graph_dat->count == display_graph_dat->granulation) {
        cur = display_graph_dat->head;
        display_graph_dat->head = display_graph_dat->head->next;
        XCopyArea(XtDisplay(w), XtWindow(w), XtWindow(w), display_graph_dat->gc,
            bar_width + 3, 0, display_graph_dat->width,  display_graph_dat->height, 0, 0);
        free(cur);
        }
    else {
        display_graph_dat->count++;
```

```
        }

    height_factor = display_graph_dat->height / display_graph_dat->scale;

    bar_value =  display_graph_dat->height - (height_factor * data);

    XSetForeground(XtDisplay(w), display_graph_dat->gc,
                                        BlackPixelOfScreen(XtScreen(w)));

    draw_filled_rectangle(w, display_graph_dat->gc, x_pos,
                                display_graph_dat->height-2,
                                x_pos+bar_width, bar_value);

    refresh(w, NULL, NULL);
}   /* draw_data  */

void resize(wgt, data, call_data)
Widget     wgt;
char       *data;
caddr_t    call_data;

/*  redraw the current screen if the window is resized  */

{
    refresh(wgt, data, call_data);
}   /* resize */


void refresh(w, data, call_data)
Widget     w;
char       *data;
caddr_t    call_data;

/*  redraw the current screen  */

{
int i, bar_value, bar_width, height_factor, x_pos;
int_node *cur;
Arg      wargs[3];

    XClearArea(XtDisplay(w), XtWindow(w), 0, 0, 0, 0, FALSE);

    XSetForeground(XtDisplay(w), display_graph_dat->gc,
                        BlackPixelOfScreen(XtScreen(w)));

    /*
    * Get the width and height that the user has set for the widget.
    */

    XtSetArg(wargs[0], XtNwidth, &display_graph_dat->width);
    XtSetArg(wargs[1], XtNheight, &display_graph_dat->height);
    XtGetValues(w, wargs,2);
```

```
    bar_width = (display_graph_dat->width -
        (display_graph_dat->granulation + 1)) / display_graph_dat->granulation;

    XSetForeground(XtDisplay(w), display_graph_dat->gc,
                        BlackPixelOfScreen(XtScreen(w)));

    for (i=0, cur = display_graph_dat->head; cur != NULL;
        cur = cur->next, i++) {
    x_pos = i * (bar_width + 1) + 2;
    height_factor = display_graph_dat->height / display_graph_dat->scale;
    bar_value =  display_graph_dat->height - (height_factor * cur->num);

    draw_filled_rectangle(w, display_graph_dat->gc, x_pos,
            display_graph_dat->height-2,
            x_pos+bar_width, bar_value);

        }
}   /* refresh */


void draw_line(w, gc, x, y, x2, y2)
Widget      w;
GC          gc;
int         x, y, x2, y2;

/* draw a single line on a graphics widget  */

{
    Display *dpy = XtDisplay(w);
    Window   win = XtWindow(w);
    XDrawLine(dpy, win, gc, x, y, x2, y2);
}    /* draw_line   */


check_points (x, y, x2, y2)
int *x, *y, *x2, *y2;

/*  figures out the right most point and assign that as x2 and y2  */

{
    if(*x2 < *x){ int tmp = *x; *x = *x2; *x2 = tmp;}
    if(*y2 < *y){ int tmp = *y; *y = *y2; *y2 = tmp;}
}   /* check_points  */


void draw_filled_rectangle(w, gc, x, y, x2, y2)
Widget      w;
GC          gc;
int         x, y, x2, y2;
{
Display *dpy = XtDisplay(w);
```

```
Window   win = XtWindow(w);

    check_points(&x, &y, &x2, &y2);
    XFillRectangle(dpy, win, gc, x, y, x2 - x, y2 - y);
}


void AddToList(w, str, pos)
Widget    w;
char      *str;
int       pos;

/*  add a new item into the scrolled list widget  */

{
 XmString xmstr;

    xmstr = XmStringCreateLtoR (str, XmSTRING_DEFAULT_CHARSET);
    XmListAddItemUnselected (w, xmstr, pos);
    XmStringFree(xmstr);
}   /* AddToList  */


ListSize (w)
Widget w;

/*  get the size of the list widget  */

{
int size;
Arg wargs[1];

    XtSetArg (wargs[0], XmNitemCount, &size);
    XtGetValues (w, wargs, 1);
    return (size);
}   /* ListSize  */


void ClearList (w)
Widget w;

/*  erase all the items on the scrolled list widget  */

{
int i, max;

    if ((max = ListSize(w)) > 0) {
        for (i = 1; i < max; i++)
            XmListDeletePos(w, LAST);
        XmListDeletePos(w, FIRST);
        }
}   /* ClearList  */
```

```
receive_from_display(head)
node_struct **head;

/* receive a linked from the display  */

{
node_struct *head1;
int n;

    read_function = display_dat->read_function;

    n = 0;
    *head = head1 = NULL;
    n = read_list(display_dat->fd_disp_out[0], &head1);
    *head = head1;
    return (n);
}   /* receive_from _display  */


send_command_to_display(head)
node_struct *head;

/*  send a linked list of command to the display process */

{
node_struct *new;

    write_function = display_dat->write_function;

    new = (node_struct *)e_malloc(sizeof(node_struct));
    new->line = (char *)e_malloc(40);
    strcpy(new->line, "_COMMAND_");
    new->next = head;
    head = new;
    write_list(display_dat->fd_disp_in[1], head);
    free(new);
}   /* receive_from_display  */


send_data_to_display(head)
node_struct *head;

/* send a linked list of data to teh display process  */

{
node_struct *new;

    write_function = display_dat->write_function;

    new = (node_struct *)e_malloc(sizeof(node_struct));
```

```
    new->line = (char *)e_malloc(10);
    strcpy(new->line, "_DATA_");
    new->next = head;
    head = new;
    write_list(display_dat->fd_disp_in[1], head);
    free(new->line);
    free(new);
}   /* send_data_to_display  */


send_block_to_display(buf, size)
char *buf;
int size;

/*  send a block of data to the display process  */

{
    write_function = display_dat->write_function;

    safe_write(display_dat->fd_disp_in[1], buf, size);
}   /* send_block_to_display */


Widget CreateFormManagedWidget(parent, name, width, height)
Widget parent;
char *name;
int width, height;

/*  create a form widget and manged it  */

{
Widget w;
Arg wargs[3];
int n = 0;

    XtSetArg(wargs[n], XmNwidth, width); n++;
    XtSetArg(wargs[n], XmNheight, height); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 0); n++;
    w = XtCreateManagedWidget(name, xmFormWidgetClass,
                                    parent, wargs, n);
    return(w);
}   /* CreateFormManagedWidget  */


Widget CreateLabel(parent, name, top_offset)
Widget parent;
char *name;
int top_offset;

/*  create a label widget  */

{
```

```
Widget w, bb;
Arg wargs[6];
int n = 0;

    XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNtopOffset, top_offset); n++;
    XtSetArg(wargs[n], XmNheight, 8); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 4); n++;
    bb = XtCreateManagedWidget ("form", xmBulletinBoardWidgetClass, parent, wargs,
                                        n);
    n = 0;
    w = XmCreateLabel(bb, name, NULL, 0);
    return(w);
}   /* CreateLabel  */


Widget CreateSingleList(parent, name, top_offset)
Widget parent;
char *name;
int top_offset;

/*  create a single item list widget  */

{
Widget w;
Arg wargs[6];
int n = 0;

    XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNtopOffset, top_offset); n++;
    XtSetArg(wargs[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 4); n++;
    w = XmCreateList(parent, name, wargs, n);
    return(w);
}   /* CreateSingleList  */


Widget CreateScrolledList(parent, name, top_offset, visible_items)
Widget parent;
char *name;
int top_offset, visible_items;

/* create a scrolled list widget  */

{
Widget w;
Arg wargs[10];
int n = 0;
```

```
    XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNtopOffset, top_offset); n++;
    XtSetArg(wargs[n], XmNselectionPolicy, XmMULTIPLE_SELECT); n++;
    XtSetArg(wargs[n], XmNscrollBarDisplayPolicy, XmSTATIC); n++;
    XtSetArg(wargs[n], XmNvisibleItemCount, visible_items); n++;
    XtSetArg(wargs[n], XmNlistSizePolicy, XmCONSTANT); n++;
    XtSetArg(wargs[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 4); n++;
    w = XmCreateScrolledList(parent, name, wargs, n);
    return(w);
}   /* CreateScrolledList  */


Widget CreateSingleLineText(parent, name, top_offset)
Widget parent;
char *name;
int top_offset;

/*  create a single line text widget  */

{
Widget w;
Arg wargs[8];
int n = 0;

    XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNtopOffset, top_offset); n++;
    XtSetArg(wargs[n], XmNeditable, False); n++;
    XtSetArg(wargs[n], XmNcursorPositionVisible, False); n++;
    XtSetArg(wargs[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 4); n++;
    XtSetArg(wargs[n], XmNautoShowCursorPosition, False); n++;
    w = (Widget)XmCreateText(parent, name, wargs, n);
    return(w);
}   /* CreateSingleLineText  */


Widget CreateScrolledText(parent, name, top_offset)
Widget parent;
char *name;
int top_offset;

/*  create a scrolled text widget */

{
Widget w;
Arg wargs[9];
int n = 0;
```

```
    XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNtopOffset, top_offset); n++;
    XtSetArg(wargs[n], XmNeditable, False); n++;
    XtSetArg(wargs[n], XmNcursorPositionVisible, False); n++;
    XtSetArg(wargs[n], XmNeditMode, XmMULTI_LINE_EDIT); n++;
    XtSetArg(wargs[n], XmNrightAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 4); n++;
    w = (Widget)XmCreateScrolledText(parent, name, wargs, n);
    return (w);
}   /* CreateScrolledText */


Widget get_menu_bar()

/* return the menubar widget to the caller */

{
    return(display_dat->menubar);
}   /* get_menu_bar */


Widget get_toplevel()

/* return the toplevel widget to the caller */

{
    return(display_dat->toplevel);
}   /* get_toplevel */


Widget get_display()

{
    return(display_list_dat->display);
}   /* get_display */


void text_snap(argc, argv)
int argc;
char *argv[];

/*  check the command line argument for the display */

{
    fprintf(stderr, "inside text_snap %s",argv[2]);
    if (strcmp(argv[1],"_TITLE") == 0) {
        XmTextSetString(display_text_dat->title, argv[2]);
        fprintf(stderr, "%s",argv[2]);
        if (strcmp(argv[3],"_DISPLAY") == 0)
            XmTextSetString(display_text_dat->display, argv[4]);
```

```
        }  /* if argv[1] == _TITLE  */
    else if (strcmp(argv[1],"_DISPLAY") == 0) {
        XmTextSetString(display_text_dat->display, argv[2]);
        if (strcmp(argv[3], "_TITLE") == 0)
            XmTextSetString(display_text_dat->title, argv[4]);
        }  /* if argv[1] == _DISPLAY */
}  /* text_snap  */
```

E.3 menus.c

```
/*************************************************************/
/*  program      : menus.c                                   */
/*  programmer   : Jeffrey Wijono                            */
/*  description  : a collection of functions for creating menus */
/*************************************************************/

#include <stdio.h>
#include "miscellaneous.h"
#include "display.h"

#include <X11/Shell.h>
#include <Xm/Xm.h>
#include <Xm/BulletinB.h>
#include <Xm/CascadeB.h>
#include <Xm/Form.h>
#include <Xm/Label.h>
#include <Xm/List.h>
#include <Xm/PushB.h>
#include <Xm/RowColumn.h>
#include <Xm/Separator.h>
#include <Xm/Text.h>
#include <Xm/ToggleBG.h>
#include <Xm/SeparatoG.h>


void    create_menu_buttons();
void    ok_save_list_callback();
void    ok_save_text_callback();
void    ok_command_callback();
void    ok_delay_callback();
void    ok_key_callback();
void    ok_col_callback();
void    ok_granula_callback();
void    ok_title_callback();
void    ok_scale_callback();
void    SetCommandHeading();
void    reset_toggle();
void    toggled();
void    reset_text_widget();
void    cancel_callback();
void    clear_callback();
```

```
void    DefaultButton();
void    clear_menu_list();


extern display_win_struct     *display_dat;
extern display_list_struct    *display_list_dat;
extern display_text_struct    *display_text_dat;
extern display_graph_struct   *display_graph_dat;
menu_struct *menu_dat;
Widget file_save=NULL, save_text=NULL, co_command=NULL, co_delay=NULL;
Widget key_select=NULL, col_select=NULL;


void initialize_menu( )

{   /* initialize_menu */
    menu_dat = NULL;
}   /* initialize_menu */


void initialize_text_menu( )

{   /* initialize_menu */
    menu_dat = NULL;
}   /* initialize_menu */


Widget CreateMenubar(display_list_dat, name)
display_list_struct *display_list_dat;
char *name;

{
Widget menubar;
Arg wargs[3];
int n = 0, nitems=0;

    XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 4); n++;
    menubar = (Widget)XmCreateMenuBar(display_dat->aForm, name, wargs, n);

    create_menu_buttons(menubar, menu_dat);
    return(menubar);
}   /* CreateMenubar */

Widget CreateTextMenubar(display_text_dat, name)
display_text_struct *display_text_dat;
char *name;

{
Widget menubar;
Arg wargs[3];
```

```
int n = 0, nitems=0;

    XtSetArg(wargs[n], XmNtopAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNshadowThickness, 4); n++;
    menubar = (Widget)XmCreateMenuBar(display_dat->aForm, name, wargs, n);
    create_menu_buttons(menubar, menu_dat);

    return(menubar);
}   /* CreateTextMenubar */


menu_struct *new_menu, *last_item=NULL;
menu_list_struct *menu_list_head;

void menu(name, func)
char    *name;
void    (* func) ();

{   /* CreatePulldownMenu */
menu_struct *cur_menu;
menu_list_struct *new_menu_list;

    new_menu      = (menu_struct *)XtMalloc(sizeof(menu_struct));
    new_menu->name  = name;
    new_menu->func  = func;
    new_menu->next  = NULL;
    new_menu->sub_menu = NULL;

    if (menu_dat == NULL) {
        menu_dat = new_menu;
        }
    else {
        if (menu_list_head == NULL) {
            for (cur_menu=menu_dat; cur_menu->next != NULL;
                cur_menu = cur_menu->next);
                cur_menu->next = new_menu;
            }
        else {
            if (last_item == menu_list_head->menu)
                last_item->sub_menu = new_menu;
            else
                last_item->next = new_menu;
            }
        }
    last_item = new_menu;
    new_menu_list = (menu_list_struct *)XtMalloc(sizeof(menu_list_struct));
    new_menu_list->next = menu_list_head;
    menu_list_head = new_menu_list;
    new_menu_list->menu = new_menu;
}   /* CreatePulldownMenu */
```

```
void menu_item (name, callback_func)
char    *name;
void    (*callback_func)();

{   /* add_menu_item */
menu_struct  *new_item;

    new_item   = (menu_struct *)XtMalloc(sizeof(menu_struct));
    new_item->name  = name;
    new_item->func  = callback_func;
    new_item->next     = NULL;
    new_item->sub_menu  = NULL;

    if (menu_list_head->menu == last_item)
        last_item->sub_menu = new_item;
    else {
        last_item->next = new_item;
        }
    last_item = new_item;
}   /* menu_item */


void done_menu()

{
menu_list_struct *prev;

    prev = menu_list_head;
    last_item = menu_list_head->menu;
    if (menu_list_head != NULL) {
        menu_list_head = menu_list_head->next;
        XtFree(prev);
        }
}   /* done_menu */


void clear_menu_list()

{
menu_list_struct *prev_list, *cur_list;

    prev_list = menu_list_head;
    for (cur_list = menu_list_head; cur_list != NULL; cur_list = cur_list->next) {
        if (prev_list != NULL)
            XtFree(prev_list);
            prev_list = cur_list;
            }
    if (prev_list != NULL)
        XtFree(prev_list);
    menu_list_head = NULL;
}   /* clear_menu_list */
```

```
void create_menu_buttons (parent, menulist)
Widget parent;
menu_struct *menulist;

/*  this is a recursive function to create the menu from the menulist */
{
Arg       wargs[1];
int       separators = 0;
Widget    button;
menu_struct *cur_item;

    for (cur_item = menulist; cur_item != NULL; cur_item = cur_item->next) {
        if (cur_item->name == NULL) {
            XtCreateManagedWidget("separator", xmSeparatorWidgetClass,
                                        parent, NULL, 0);
            }
        else if (cur_item->sub_menu) {
            Widget sub_menu;
            sub_menu = XmCreatePulldownMenu(parent, menulist->name, NULL, 0);
            XtSetArg(wargs[0], XmNsubMenuId, sub_menu);
            button = XtCreateManagedWidget(cur_item->name,
                        xmCascadeButtonWidgetClass, parent, wargs, 1);
            create_menu_buttons(sub_menu, cur_item->sub_menu);
            }
        else {
            button = XtCreateManagedWidget(cur_item->name,
                        xmPushButtonWidgetClass, parent, NULL, 0);
            XtAddCallback(button, XmNactivateCallback, cur_item->func, NULL);
            }
        }   /* for cur_menu != NULL */
}           /* create_menu_buttons  */


void DoSaveList()

/*  save the current list display content to a file  */

{
Widget label, ok_button, cancel_button, clear_button;
Arg   wargs[10];
int   n;
dialog_data_struct *d_data;

if (file_save == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 1;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 125); n++;
```

```
XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
d_data->label = file_save =
XmCreateFormDialog(display_dat->menubar, "File Save", wargs, n);
/*
* Create one single line XmEdit widgets
* and associate a button with each text widget.
* Assign an XmNactivateCallback callback to each button.
*/
n = 0;
XtSetArg(wargs[n], XmNx, 10); n++;
XtSetArg(wargs[n], XmNy, 40); n++;
d_data->value[0] = XtCreateManagedWidget("field1", xmTextWidgetClass,
            d_data->label, wargs,n);
d_data->value[1] = display_list_dat->title;
d_data->value[2] = display_list_dat->display;
n = 0;
XtSetArg(wargs[n], XmNx, 10); n++;
XtSetArg(wargs[n], XmNy, 10); n++;
label = XtCreateManagedWidget("Enter File Name:",
            xmLabelWidgetClass, d_data->label, wargs,n);

n = 0;
XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftOffset, 10); n++;
XtSetArg(wargs[n], XmNheight, 33); n++;
ok_button= XtCreateManagedWidget(" OK ",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);

XtAddCallback(ok_button, XmNactivateCallback,
            ok_save_list_callback, d_data);
DefaultButton(d_data->label, ok_button);

n = 0;
XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftOffset, 66); n++;
XtSetArg(wargs[n], XmNheight, 33); n++;
cancel_button= XtCreateManagedWidget("CANCEL",
                    xmPushButtonWidgetClass,
                    d_data->label, wargs, n);
XtAddCallback(cancel_button, XmNactivateCallback,
        cancel_callback, d_data);
n = 0;
XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftOffset, 130); n++;
XtSetArg(wargs[n], XmNheight, 33); n++;
```

```
    clear_button= XtCreateManagedWidget("CLEAR",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
    clear_callback, d_data);
    }
XtManageChild(file_save);
}   /* DoSaveList */


void ok_save_list_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;
{
XmStringTable table;
XmString xmstr;
Arg wargs[2];
int n, count;
char str[30], *line;
FILE *fptr;

    XtUnmanageChild(dialog->label);
    strcpy(str, XmTextGetString(dialog->value[0]));
    fptr = fopen(str,"w");

    XtSetArg(wargs[0], XmNlabelString, &xmstr);
    XtGetValues(dialog->value[1], wargs, 1);
    XmStringGetLtoR(xmstr, XmSTRING_DEFAULT_CHARSET, &line);
    fprintf(fptr,"%s\n",line);
    XtFree(line);
    XtSetArg(wargs[0], XmNitems, &table);
    XtSetArg(wargs[1], XmNitemCount, &count);
    XtGetValues(dialog->value[2], wargs, 2);
    for (n=0; n < count; n++) {
        XmStringGetLtoR(table[n], XmSTRING_DEFAULT_CHARSET, &line);
        fprintf(fptr,"%s\n",line);
    XtFree(line);
    }

    reset_text_widget(dialog->value[0]);
    fclose(fptr);
}   /* ok_save_list_callback  */


void DoSaveText()

/*  save the current text display content to a file  */

{
Widget label, ok_button, cancel_button, clear_button;
Arg    wargs[10];
```

```
int   n;
dialog_data_struct *d_data;

if (save_text == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 1;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 125); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    d_data->label = save_text = XmCreateFormDialog(display_dat->menubar, "File Save",
wargs, n);
    /*
    * Create one single line XmEdit widgets
    * and associate a button with each text widget.
    * Assign an XmNactivateCallback callback to each button.
    */
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 40); n++;
    d_data->value[0] = XtCreateManagedWidget("field1", xmTextWidgetClass,
                d_data->label, wargs,n);
    d_data->value[1] = display_list_dat->title;
    d_data->value[2] = display_list_dat->display;
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 10); n++;
    label = XtCreateManagedWidget("Enter File Name:",
                        xmLabelWidgetClass,
                        d_data->label, wargs,n);

    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 10); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    ok_button= XtCreateManagedWidget(" OK ",
                            xmPushButtonWidgetClass,
                            d_data->label, wargs, n);

    XtAddCallback(ok_button, XmNactivateCallback,
                ok_save_text_callback, d_data);
    DefaultButton(d_data->label, ok_button);

    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 66); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
```

```
    cancel_button= XtCreateManagedWidget("CANCEL",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(cancel_button, XmNactivateCallback,
            cancel_callback, d_data);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 130); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    clear_button= XtCreateManagedWidget("CLEAR",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
            clear_callback, d_data);
    }
XtManageChild(save_text);
}   /* DoSaveText */


void ok_save_text_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
char str[30];
FILE *fptr;

    strcpy(str, XmTextGetString(dialog->value[0]));
    fptr = fopen(str,"w");
    XtUnmanageChild(dialog->label);
    fputs(XmTextGetString(dialog->value[1]),fptr);
    fputs("\n",fptr);
    fputs(XmTextGetString(dialog->value[2]),fptr);

    reset_text_widget(dialog->value[0]);
    fclose(fptr);
}   /* ok_save_text_callback */


void DoPrintList()

/*  prints the current list display content to a standard printer  */

{
char *str;
int n, count;
XmStringTable table;
XmString xmstr;
Arg wargs[2];
```

```
FILE *fptr;

    fptr = fopen("print.tmp.file","w");
    XtSetArg(wargs[0], XmNlabelString, &xmstr);
    XtGetValues(display_list_dat->title, wargs, 1);
    XmStringGetLtoR(xmstr, XmSTRING_DEFAULT_CHARSET, &str);
    fprintf(fptr,"%s\n",str);
    XtFree(str);
    XtSetArg(wargs[0], XmNitems, &table);
    XtSetArg(wargs[1], XmNitemCount, &count);
    XtGetValues(display_list_dat->display, wargs, 2);
    for (n=0; n < count; n++) {
        XmStringGetLtoR(table[n], XmSTRING_DEFAULT_CHARSET, &str);
        fprintf(fptr,"%s\n",str);
        XtFree(str);
        }
    fclose(fptr);
    system("lpr print.tmp.file");
}   /* DoPrintList */


void DoPrintText()

/*  prints the current text display content to a standard printer  */
{
FILE *fptr;

    fptr = fopen("print.tmp.file","w");
    fputs(XmTextGetString(display_list_dat->title), fptr);
    fputs("\n",fptr);
    fputs(XmTextGetString(display_list_dat->display), fptr);
    fclose(fptr);
    system("lpr print.tmp.file");
}   /* DoPrintText */


void DoQuit(w, file_quit, call_data)
Widget w;
caddr_t *file_quit;
XmAnyCallbackStruct call_data;

/*  sends the terminate command to the main program  */

{
node_struct *head;

    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = "_TERMINATE";
    write_list(display_dat->fd_disp_out[1], head);
    XtFree(head);
}   /* DoQuit */
```

```
void DoQuit2(w, file_quit, call_data)
Widget w;
caddr_t *file_quit;
XmAnyCallbackStruct call_data;

{
    XtCloseDisplay(XtDisplay(display_dat->aForm));
    exit(0);
}   /* DoQuit2 */


void DoUndo()

/*  sends the undo command to the main programm */

{
node_struct *head;

    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = "_UNDO";
    write_list(display_dat->fd_disp_out[1], head);
    XtFree(head);
}


void DoCommand()

/*  gets the new command to be processed from the text widget and
    sneds it to the main program  */

{
Widget label, toggle, ok_button, cancel_button, clear_button;
Arg    wargs[10];
int    n;
dialog_data_struct *d_data;

if (co_command == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 1;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 150); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    co_command = d_data->label =
        XmCreateFormDialog(display_dat->menubar, "Command", wargs, n);
    /*
    * Create one single line XmEdit widgets
    * and associate a button with each text widget.
```

```
* Assign an XmNactivateCallback callback to each button.
*/
n = 0;
XtSetArg(wargs[n], XmNx, 10); n++;
XtSetArg(wargs[n], XmNy, 40); n++;
d_data->value[0] = XtCreateManagedWidget("field1",
            xmTextWidgetClass,
            d_data->label, wargs,n);
n = 0;
XtSetArg(wargs[n], XmNx, 10); n++;
XtSetArg(wargs[n], XmNy, 10); n++;
label = XtCreateManagedWidget("UNIX Command:", xmLabelWidgetClass,
                                    d_data->label, wargs, n);

n = 0;
XtSetArg(wargs[n], XmNx, 10); n++;
XtSetArg(wargs[n], XmNy, 72); n++;
XtSetArg(wargs[n], XmNindicatorOn, True); n++;
toggle = XtCreateManagedWidget("Command Without Heading",
                    xmToggleButtonGadgetClass,
                    d_data->label, wargs, n);
XtAddCallback(toggle, XmNvalueChangedCallback, SetCommandHeading, NULL);

n = 0;
XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftOffset, 10); n++;
XtSetArg(wargs[n], XmNheight, 30); n++;
ok_button= XtCreateManagedWidget(" OK ", xmPushButtonWidgetClass,
                d_data->label, wargs, n);
XtAddCallback(ok_button, XmNactivateCallback, ok_command_callback, d_data);
DefaultButton(d_data->label, ok_button);
n = 0;
XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftOffset, 66); n++;
XtSetArg(wargs[n], XmNheight, 30); n++;
cancel_button= XtCreateManagedWidget("CANCEL",
                    xmPushButtonWidgetClass,
                    d_data->label, wargs, n);
XtAddCallback(cancel_button, XmNactivateCallback,
                    cancel_callback, d_data);
n = 0;
XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
XtSetArg(wargs[n], XmNleftOffset, 130); n++;
XtSetArg(wargs[n], XmNheight, 30); n++;
clear_button= XtCreateManagedWidget("CLEAR",
                    xmPushButtonWidgetClass,
```

```
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
                        clear_callback, d_data);
    }   /* if co_command == NULL */
    XtManageChild(co_command);
}   /* DoCommand    */


void   SetCommandHeading(w, data, toggle_data)
Widget w;
XmAnyCallbackStruct *data;
XmToggleButtonCallbackStruct *toggle_data;

{
int heading;

    if (toggle_data->set == True)
        heading = 1;
    else
        heading = 0;
}   /* SetCommandHeading */


void ok_command_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
node_struct *head, *new;

    XtUnmanageChild(dialog->label);
    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->line = (char *)XtMalloc(20);
    strcpy(head->line, "_CHANGE_COMMAND");
    new = (node_struct *)XtMalloc(sizeof(node_struct));
    new->next = NULL;
    new->line = (char *)XtMalloc(40);
    strcpy(new->line, XmTextGetString(dialog->value[0]));
    head->next = new;
    write_list(display_dat->fd_disp_out[1], head);
    reset_text_widget(dialog->value[0]);
    clear_node(&head);
}   /* ok_command_callback */


void DoDelay()

/*  gets the new number of the duratrion time from the text widget
    ans sends it to the main program  */

{
```

```
Widget label, ok_button, cancel_button, clear_button;
Arg   wargs[10];
int   n;
dialog_data_struct *d_data;

if (co_delay == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 1;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 125); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    co_delay = d_data->label
        = XmCreateFormDialog(display_dat->menubar, "Duration", wargs, n);
    /*
    * Create one single line XmEdit widgets
    * and associate a button with each text widget.
    * Assign an XmNactivateCallback callback to each button.
    */
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 40); n++;
    d_data->value[0] = XtCreateManagedWidget("field1",
                        xmTextWidgetClass,
                        d_data->label, wargs,n);
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 10); n++;
    label = XtCreateManagedWidget("DURATION:",
                        xmLabelWidgetClass,
                        d_data->label, wargs,n);

    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 10); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    ok_button= XtCreateManagedWidget(" OK ",
                            xmPushButtonWidgetClass,
                            d_data->label, wargs, n);
    XtAddCallback(ok_button, XmNactivateCallback,
                        ok_delay_callback, d_data);
    DefaultButton(d_data->label, ok_button);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 66); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    cancel_button= XtCreateManagedWidget("CANCEL",
```

```
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(cancel_button, XmNactivateCallback,
                cancel_callback, d_data);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 130); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    clear_button= XtCreateManagedWidget("CLEAR",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
            clear_callback, d_data);
    }   /* if co_delay == NULL */
    XtManageChild(co_delay);
}   /* DoDelay */


void ok_delay_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
node_struct *head, *new;

    XtUnmanageChild(dialog->label);
    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = (char *)XtMalloc(15);
    strcpy(head->line, "_CHANGE_DELAY");
    new = (node_struct *)XtMalloc(sizeof(node_struct));
    new->next = NULL;
    new->line = (char *)XtMalloc(10);
    strcpy(new->line, XmTextGetString(dialog->value[0]));
    if (strlen(new->line) == 0 || atoi(new->line) < MIN_DELAY)
        sprintf(new->line,"%d",MIN_DELAY);
    head->next = new;
    write_list(display_dat->fd_disp_out[1], head);
    reset_text_widget(dialog->value[0]);
    clear_node(&head);
}   /* ok_delay_callback */


void DoPause()

/*  send a resume command to the main program  */

{
node_struct *head;
```

```
    display_list_dat->pause = True;
    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = "_PAUSE";
    write_list(display_dat->fd_disp_out[1], head);
  XtFree(head);
}   /* DoPause  */

void DoResume()

/*  send a pause command to the main program  */

{
node_struct *head;
Arg wargs[1];

    XmListDeselectAllItems(display_list_dat->display);
    XtSetArg(wargs[0], XmNvalue, 0);
    XtSetValues(display_list_dat->vscroll, wargs, 1);
    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = "_RESUME";
    display_list_dat->pause = False;
    write_list(display_dat->fd_disp_out[1], head);
    XtFree(head);
}   /* DoResume */

void DoKillProcess()

/*  get the selected item position number number and send these
    numbers o the main program  */

{
int n, **position_list, *count;
node_struct *head, *new;

    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = (char *)XtMalloc(15);
    strcpy(head->line, "_KILL_PROCESS");
    if (XmListGetSelectedPos(display_list_dat->display, position_list, count)) {
        for (n=0; n < *count; n++) {
            new = (node_struct *)XtMalloc(sizeof(node_struct));
            new->next = NULL;
            new->line = (char *)e_malloc(10);
            sprintf(new->line, "%d", *((*position_list)+n));
            insert_node(&head, new);
            }
        write_list(display_dat->fd_disp_out[1], head);
        }
    DoResume();
```

```
    clear_node(&head);
}   /* DoKillProcess */


void DoKeySelect()

/*  gets the keyword from the text widget and sends it to the\
    main program  */
{
Widget label[2], ok_button, cancel_button, clear_button;
Arg   wargs[10];
int   n;
dialog_data_struct *d_data;

if (key_select == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 2;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 210); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    key_select = d_data->label = XmCreateFormDialog(display_dat->aForm, "Key", wargs,
n);
    /*
    * Create one single line XmEdit widgets
    * and associate a button with each text widget.
    * Assign an XmNactivateCallback callback to each button.
    */

    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 40); n++;
    d_data->value[0] = XtCreateManagedWidget("field1",
                    xmTextWidgetClass,
                    d_data->label, wargs,n);
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 10); n++;
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 120); n++;
    d_data->value[1] = XtCreateManagedWidget("field2",
                    xmTextWidgetClass,
                    d_data->label, wargs,n);
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 10); n++;
    label[0] = XtCreateManagedWidget("ENTER KEY:",
                    xmLabelWidgetClass,
                    d_data->label, wargs,n);
    n = 0;
```

```
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 80); n++;
    label[1] = XtCreateManagedWidget("ENTER COLUMN NUMBER:",
                    xmLabelWidgetClass,
                    d_data->label, wargs,n);

    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 10); n++;
    XtSetArg(wargs[n], XmNheight, 30); n++;
    ok_button= XtCreateManagedWidget("SELECT",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(ok_button, XmNactivateCallback,
    ok_key_callback, d_data);
    DefaultButton(d_data->label, ok_button);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 70); n++;
    XtSetArg(wargs[n], XmNheight, 30); n++;
    cancel_button= XtCreateManagedWidget("CANCEL",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(cancel_button, XmNactivateCallback,
                        cancel_callback, d_data);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 130); n++;
    XtSetArg(wargs[n], XmNheight, 30); n++;
    clear_button= XtCreateManagedWidget("CLEAR",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
                        clear_callback, d_data);
    }
    XtManageChild(key_select);
}   /* DoKeySelect */


void ok_key_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
node_struct *head, *new, *new2;
```

```
    XtUnmanageChild(dialog->label);
    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = (char *)XtMalloc(15);
    strcpy(head->line, "_CHANGE_KEY");
    new = (node_struct *)XtMalloc(sizeof(node_struct));
    new->line = (char *)XtMalloc(10);
    strcpy(new->line, XmTextGetString(dialog->value[0]));
    head->next = new;
    new2 = (node_struct *)XtMalloc(sizeof(node_struct));
    new2->next = NULL;
    new2->line = (char *)XtMalloc(10);
    strcpy(new2->line, XmTextGetString(dialog->value[1]));
    if (strlen(new2->line) == 0 || strlen(new->line) == 0)
    strcpy(new2->line, "0");
    new->next = new2;
    write_list(display_dat->fd_disp_out[1], head);
    reset_text_widget(dialog->value[0]);
    reset_text_widget(dialog->value[1]);
    clear_node(&head);
}   /* ok_key_callback */


unsigned long toggles_set;

void DoColSelect()

{
Widget ok_button, cancel_button, w, check_box;
Arg    wargs[10];
int    n, i;
dialog_data_struct *d_data;
XmString xmstr;
char *str, title[MAX_COLUMN][40];
char *toggle_box[] =
{"UID","PID","PPID","C","STIME","TTY","TIME","COMMAND"};

if (col_select == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 0;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 170); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    col_select = d_data->label =
        XmCreateFormDialog(display_dat->aForm, "Column", wargs, n);

    n = 0;
    XtSetArg(wargs[n], XmNpacking, XmPACK_COLUMN); n++;
    XtSetArg(wargs[n], XmNnumColumns, 2); n++;
```

```
    check_box = XtCreateManagedWidget("check_box",
                        xmRowColumnWidgetClass,
                        d_data->label, wargs,n);
    XtGetValues(display_list_dat->title, wargs, 1);
    XmStringGetLtoR(xmstr, XmSTRING_DEFAULT_CHARSET, &str);
    /* buf_to_array(str, toggle_box); */

    n = 0;
    XtSetArg(wargs[n], XmNindicatorOn, True); n++;
    for(i=0; i < XtNumber(toggle_box); i++) {
    w = XtCreateManagedWidget(toggle_box[i],
        xmToggleButtonGadgetClass, check_box, wargs, n);
    XtAddCallback(w, XmNvalueChangedCallback, toggled, i);
    }
    XtCreateManagedWidget("_sep",
        xmSeparatorGadgetClass, d_data->label, NULL, 0);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 10); n++;
    XtSetArg(wargs[n], XmNheight, 30); n++;
    d_data->num_of_col = XtNumber(toggle_box);
    ok_button= XtCreateManagedWidget(" OK ",
                    xmPushButtonWidgetClass,
                    d_data->label, wargs, n);
    XtAddCallback(ok_button, XmNactivateCallback,
            ok_col_callback, d_data);
    DefaultButton(d_data->label, ok_button);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 80); n++;
    XtSetArg(wargs[n], XmNheight, 30); n++;
    cancel_button= XtCreateManagedWidget("CANCEL",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(cancel_button, XmNactivateCallback,
            cancel_callback, d_data);
    }
    XtManageChild(col_select);
}


void toggled(w, bit, toggle_data)
Widget w;
int   bit;
XmToggleButtonCallbackStruct *toggle_data;
{
    if (toggle_data->set)        /* if the toggle button is set, flip its bit */
        toggles_set |= (1 << bit);
```

```
    else                        /* if the toggle is "off", turn off the bit. */
        toggles_set &= ~(1 << bit);
}   /* toggles */


void ok_col_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
int i;
char tmp[5];
node_struct *head, *new;

    XtUnmanageChild(dialog->label);
    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->line = (char *)XtMalloc(15);
    strcpy(head->line, "_COL_SELECT");
    new = (node_struct *)XtMalloc(sizeof(node_struct));
    new->line = (char *)XtMalloc(30);
    strcpy(new->line, "\0");
    new->next = NULL;
    for (i=0; i < dialog->num_of_col; i++)
        if (toggles_set & (1<<i)) {
            sprintf(tmp,"%d ",i+1);
            strcat(new->line, tmp);
            }
    head->next = new;
    write_list(display_dat->fd_disp_out[1], head);
    clear_node(&head);
}   /* ok_col_callback */


void DoRowSelect()

/*  get the selection of the list widget and sends it to the
    main program    */

{
int n, **position_list, *count;
node_struct *head, *new;

    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = (char *)XtMalloc(15);
    strcpy(head->line, "_ROW_SELECT");
    if (XmListGetSelectedPos(display_list_dat->display, position_list, count)) {
        for (n=0; n < *count; n++) {
            new = (node_struct *)XtMalloc(sizeof(node_struct));
            new->next = NULL;
            new->line = (char *)e_malloc(10);
```

```
            sprintf(new->line, "%d", *((*position_list)+n));
            insert_node(&head, new);
            }
        write_list(display_dat->fd_disp_out[1], head);
        }
    DoResume();
    clear_node(&head);
}   /* DoRowSelect  */


void UndoRowSelect()

/*  Undo the row selection of the list widget  */

{
node_struct *head;

    head = (node_struct *)XtMalloc(sizeof(node_struct));
    head->next = NULL;
    head->line = (char *)XtMalloc(20);
    strcpy(head->line, "_UNDO_ROW_SELECT");
    write_list(display_dat->fd_disp_out[1], head);
    clear_node(&head);
}   /* UndoRowSelect  */


void DoSnap()

/* create another process to capture the current screen  */

{
char *str, *title, *buf2;
int c, n, count;
XmStringTable table;
XmString xmstr;
Arg wargs[2];

    if (c = fork(), c == 0) {
        XtSetArg(wargs[0], XmNlabelString, &xmstr);
        XtGetValues(display_list_dat->title, wargs, 1);
        XmStringGetLtoR(xmstr, XmSTRING_DEFAULT_CHARSET, &title);
        title[strlen(title)] = '\0';
        XtSetArg(wargs[0], XmNitems, &table);
        XtSetArg(wargs[1], XmNitemCount, &count);
        XtGetValues(display_list_dat->display, wargs, 2);
        buf2 = (char *)e_malloc(SNAP_SIZE);
        for (n=0; n < count; n++) {
            XmStringGetLtoR(table[n], XmSTRING_DEFAULT_CHARSET, &str);
            strcat(buf2, str);
            strcat(buf2, "\n");
            XtFree(str);
            }
```

```
        buf2[strlen(buf2)] = '\0';
        execl("./snap", "snap", "_TITLE", title, "_DISPLAY", buf2, NULL);
        free(buf2);
        }
    else if (c < 0)
        syserr("fork fail at snap shot");
}   /*  DoSnap  */


Widget granula = NULL, scale = NULL;

void DoGranulation()

/* allow user to set the granulation on the global variable  */
{
Widget label, ok_button, cancel_button, clear_button;
Arg    wargs[10];
int    n;
dialog_data_struct *d_data;

if (granula == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 1;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 125); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    granula = d_data->label
        = XmCreateFormDialog(display_dat->menubar, "Duration", wargs, n);
    /*
    * Create one single line XmEdit widgets
    * and associate a button with each text widget.
    * Assign an XmNactivateCallback callback to each button.
    */
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 40); n++;
    d_data->value[0] = XtCreateManagedWidget("field1",
                xmTextWidgetClass,
                d_data->label, wargs,n);
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 10); n++;
    label = XtCreateManagedWidget("GRANULATION:",
                    xmLabelWidgetClass,
                    d_data->label, wargs,n);

    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
```

```
    XtSetArg(wargs[n], XmNleftOffset, 10); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    ok_button= XtCreateManagedWidget(" OK ",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(ok_button, XmNactivateCallback,
            ok_granula_callback, d_data);
    DefaultButton(d_data->label, ok_button);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 66); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    cancel_button= XtCreateManagedWidget("CANCEL",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(cancel_button, XmNactivateCallback,
                        cancel_callback, d_data);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 130); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    clear_button= XtCreateManagedWidget("CLEAR",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
    clear_callback, d_data);
    }   /* if co_delay == NULL */
XtManageChild(granula);
}   /* DoGranulation  */


void ok_granula_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
char *str, *buf;

    XtUnmanageChild(dialog->label);

    display_graph_dat->granulation = atoi(XmTextGetString(dialog->value[0]));
    reset_text_widget(dialog->value[0]);
}   /* ok_granulation  */


Widget set_title_widget = NULL;
```

```
void DoSetTitle()

/* set the title on the title widget */

{
Widget label, ok_button, cancel_button, clear_button;
Arg   wargs[10];
int   n;
dialog_data_struct *d_data;

if (set_title_widget == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 1;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 125); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    set_title_widget = d_data->label
        = XmCreateFormDialog(display_dat->menubar, "Duration", wargs, n);
    /*
    * Create one single line XmEdit widgets
    * and associate a button with each text widget.
    * Assign an XmNactivateCallback callback to each button.
    */
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 40); n++;
    d_data->value[0] = XtCreateManagedWidget("field1",
                        xmTextWidgetClass,
                        d_data->label, wargs,n);
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 10); n++;
    label = XtCreateManagedWidget("NEW TITLE:",
                        xmLabelWidgetClass,
                        d_data->label, wargs,n);

    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 10); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    ok_button= XtCreateManagedWidget(" OK ",
                            xmPushButtonWidgetClass,
                            d_data->label, wargs, n);
    XtAddCallback(ok_button, XmNactivateCallback,
                            ok_title_callback, d_data);
    DefaultButton(d_data->label, ok_button);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
```

```
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 66); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    cancel_button= XtCreateManagedWidget("CANCEL",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(cancel_button, XmNactivateCallback,
                        cancel_callback, d_data);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 130); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    clear_button= XtCreateManagedWidget("CLEAR",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
                        clear_callback, d_data);
    }   /* if co_delay == NULL */
XtManageChild(set_title_widget);
}   /* DoSetTitle  */


void ok_title_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;
{
char *str, *buf;
Arg wargs[1];
XmString xmstr;

    XtUnmanageChild(dialog->label);

    strcpy(display_graph_dat->title_name,
                XmTextGetString(dialog->value[0]));
    xmstr = XmStringCreateLtoR(display_graph_dat->title_name,
                XmSTRING_DEFAULT_CHARSET);
    XtSetArg(wargs[0], XmNlabelString, xmstr);
    XtSetValues(display_graph_dat->title, wargs, 1);
    XtFree(xmstr);

    reset_text_widget(dialog->value[0]);
}   /* ok_title  */


void DoScale()

{
Widget label, ok_button, cancel_button, clear_button;
```

```
Arg   wargs[10];
int   n;
dialog_data_struct *d_data;

if (scale == NULL) {
    d_data = (dialog_data_struct *)XtMalloc(sizeof(dialog_data_struct));
    d_data->value[0] = d_data->value[1] = d_data->value[2] = NULL;
    d_data->type = 1;
    n = 0;
    XtSetArg(wargs[n], XmNwidth, 200); n++;
    XtSetArg(wargs[n], XmNheight, 125); n++;
    XtSetArg(wargs[n], XmNautoUnmanage, FALSE); n++;
    scale = d_data->label
    = XmCreateFormDialog(display_dat->menubar, "Duration", wargs, n);
    /*
    * Create one single line XmEdit widgets
    * and associate a button with each text widget.
    * Assign an XmNactivateCallback callback to each button.
    */
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 40); n++;
    d_data->value[0] = XtCreateManagedWidget("field1",
                xmTextWidgetClass,
                d_data->label, wargs,n);
    n = 0;
    XtSetArg(wargs[n], XmNx, 10); n++;
    XtSetArg(wargs[n], XmNy, 10); n++;
    label = XtCreateManagedWidget("Height Scale:",
                    xmLabelWidgetClass,
                    d_data->label, wargs,n);

    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 10); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    ok_button= XtCreateManagedWidget(" OK ",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(ok_button, XmNactivateCallback,
                        ok_scale_callback, d_data);
    DefaultButton(d_data->label, ok_button);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 66); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    cancel_button= XtCreateManagedWidget("CANCEL",
                        xmPushButtonWidgetClass,
```

```
                        d_data->label, wargs, n);
    XtAddCallback(cancel_button, XmNactivateCallback,
                        cancel_callback, d_data);
    n = 0;
    XtSetArg(wargs[n], XmNbottomAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNbottomOffset, 10); n++;
    XtSetArg(wargs[n], XmNleftAttachment, XmATTACH_FORM); n++;
    XtSetArg(wargs[n], XmNleftOffset, 130); n++;
    XtSetArg(wargs[n], XmNheight, 33); n++;
    clear_button= XtCreateManagedWidget("CLEAR",
                        xmPushButtonWidgetClass,
                        d_data->label, wargs, n);
    XtAddCallback(clear_button, XmNactivateCallback,
                        clear_callback, d_data);
    }   /* if co_delay == NULL */
XtManageChild(scale);
}   /*  DoScale  */


void ok_scale_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
node_struct *head, *new;

    XtUnmanageChild(dialog->label);

    display_graph_dat->scale = atoi(XmTextGetString(dialog->value[0]));
    reset_text_widget(dialog->value[0]);
}


void cancel_callback(w, dialog, call_data)
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{

    XtUnmanageChild(dialog->label);
    if (dialog->type == 1)
        reset_text_widget(dialog->value[0]);
    else if (dialog->type == 2) {
        reset_text_widget(dialog->value[0]);
        reset_text_widget(dialog->value[1]);
        }
}   /* cancel_callback */


void clear_callback(w, dialog, call_data)
```

```
Widget    w;
dialog_data_struct *dialog;
XmAnyCallbackStruct *call_data;

{
    if (dialog->type == 1)
        reset_text_widget(dialog->value[0]);
    else if (dialog->type == 2) {
        reset_text_widget(dialog->value[0]);
        reset_text_widget(dialog->value[1]);
        }
}   /* clear_callback  */


void reset_text_widget(w)
Widget w;

{
    XmTextSetString(w,"");
    XmTextSetInsertionPosition(w,0);
}  /*  reset_text_widget  */


void DefaultButton (parent, w)
Widget parent,w;

{
    Arg wargs[1];

    XtSetArg(wargs[0], XmNdefaultButton, w);
    XtSetValues(parent, wargs, 1);
}   /* DefaultButton  */
```

Appendix F

Miscellaneous Library's Code

F.1 miscellaneous.h

```
/*************************************************************/
/* Program      : miscellaneous.h                            */
/* Programmer   : Jeffrey Wijono                             */
/* description  : This file contains header file for miscellaneous library */
/*************************************************************/

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>

#define COL_WIDTH       90
#define MAX_COLUMN       8
#define STD_INPUT        0
#define STD_OUTPUT       1
#define MIN_DELAY        1
#define FALSE            0
#define TRUE             1
#define PIPE             1
#define SHARE_MEMORY     2

typedef struct node {
        char            *line;          /* data line */
        struct node     *next;          /* pointer to next node struct */
        } node_struct;

typedef struct _int {
        int             num;            /* integer node */
        struct _int     *next;          /* pointer to the next int_node struct */
        } int_node;

void    insert_node();
void    clear_node();
void    insert_int_node();
void    clear_int_node();
char    *e_malloc();
```

F.2 Miscellaneous.c

```
/*************************************************************/
/*  program     : miscellaneous.c                            */
/*  programmer  : Jeffrey Wijono                             */
/*  description : contains miscellaneous functions           */
/*************************************************************/
```

```
#include "miscellaneous.h"

min(a, b)
int a, b;

{
    if(a < b) return(a);
    return(b);
}


syserr(msg)
char *msg;

/*  display the error message and the error number when
    the error is occured */

{
extern int errno, sys_nerr;
extern char *sys_errlist[];

    fprintf(stderr,"ERROR: %s (%d", msg, errno);
    if (errno > 0 && errno < sys_nerr)
        fprintf(stderr, "; %s)\n", sys_errlist[errno]);
    else
        fprintf(stderr, ")\n");
    exit(1);
}


char *e_malloc(size)
int size;

{
char *buf;

    if ((buf = (char *)malloc(size)) == NULL) {
        fprintf(stderr, "FAIL on malloc size %d\n", size);
        exit(0);
        }
    return(buf);
}


void insert_node(head, new)
node_struct **head, *new;

{
node_struct *cur;
```

```
    if (*head == NULL)
        *head = new;
    else {
        for (cur = *head; cur->next != NULL; cur = cur->next);
        cur->next = new;
    }
}   /*  insert_node  */


void clear_node(head)
node_struct **head;

/* empty the list and point the head of the list to NULL */

{
node_struct *cur, *prev;

    prev = NULL;
    for (cur = *head; cur != NULL; ) {
        prev = cur;
        cur = cur->next;
        free(prev->line);
        free(prev);
        }
    *head = NULL;
}   /*   clear_node  */


void insert_int_node(head, pid)
int_node **head;
int pid;

{
int_node *new, *cur;

    new = (int_node *)e_malloc(sizeof(int_node));
    new->num = pid;
    new->next = NULL;
    if (*head == NULL)
        *head = new;
    else {
        for (cur = *head; cur->next != NULL; cur = cur->next);
        cur->next = new;
        }
}   /* insert_int_node */


void clear_int_node(head)
int_node **head;

/* empty the list and point the head of the list to NULL */
```

```
{
int_node *cur, *prev;

    prev = NULL;
    for (cur = *head; cur != NULL; ) {
        prev = cur;
        cur = cur->next;
        free(prev);
        }
    *head = NULL;
}   /* clear_int_node  */


insert_int_str(head, str)
int_node **head;
char *str;

{
int_node *new, *cur;

    new = (int_node *)e_malloc(sizeof(int_node));
    new->num = atoi(str);
    new->next = NULL;
    if (*head == NULL)
        *head = new;
    else {
        for (cur = *head; cur->next != NULL; cur = cur->next);
        cur->next = new;
        }
}


is_command(head, command)
node_struct *head;
char *command;

{
    return(strcmp(head->line, command));
}


set_file_descriptor_no_delay(fd1)
int fd1[2];

{
    fcntl(fd1[0], F_SETFL, O_NDELAY);
    fcntl(fd1[1], F_SETFL, O_NDELAY);
}


buf_to_array(str, buf_array)
char *str, *buf_array[];
```

```
{
char *p, *temp;
int n;

    for (n=0; n<=MAX_COLUMN; n++)
        buf_array[n][0] = '\0';
    n=0;
    temp = (char *)malloc(strlen(str)+1);
    strcpy(temp, str);
    p = strtok(temp, " ");
    do {
        strcpy(buf_array[n], p);
        p = strtok('\0', ", ");
        n++;
        } while (p);
}    /* buf_to_array */
```