NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Developing
Portable User Interfaces
for Ada Command Control Software

by

Chien Hsiung Sun

June 1990

Thesis Advisor: Luqi

Approved for public release; distribution is unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable): 52	7a. NAME OF MONITORING ORGANIZATION: Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code): Monterey, CA 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
DEVELOPING PORTABLE USER INTERFACES FOR ADA COMMAND CONTROL SOFTWARE

12. PERSONAL AUTHOR(S)
Sun, Chien Shiung

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM 01/88 TO 06/90	June 1990	135

16. SUPPLEMENTARY NOTATION: The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. COSATI CODES		
FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
To find an efficient asynchronous interface between X-Windows and real-time Ada programs.

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

DoD mandated use of Ada for embedded systems includes Combat and Control System such as shipboard Combat Direction System (CDS). A Low Cost CDS (LCCDS) which will use commercial workstation platforms will require interfacing Ada real-time programs with a portable windowing environment such as X Windows. This thesis explore several methods for building X Windows based user interface for Ada C^2 programs and provides a step-by-step approach to user interface design for future CDS developers.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Luqi	22b. TELEPHONE (Include Area Code): (408) 646-2735; 22c. OFFICE SYMBOL: 52Lq

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Approved for public release; distribution is unlimited

Developing Portable User Interfaces for Ada Command Control Software

by

Chien Hsiung Sun
LCDR, Taiwan navy
B.S., Taiwan Naval Academy, 1979

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF ENGINEERING SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: Chien Hsiung Sun

Approved By: Luqi, Thesis Advisor

Captain Patrick D. Barnes, Second Reader

Robert B. McGhee, Chairman,
Department of Computer Science

ABSTRACT

DoD mandated use of Ada for embedded systems includes Combat and Control System such as shipboard Combat Direction System (CDS). A Low Cost CDS (LCCDS) which will use commercial workstation platforms will require interfacing Ada real-time programs with a portable windowing environment such as X Windows. This thesis explore several methods for building X Windows based user interface for Ada C^2 programs and provides a step-by-step approach to user interface design for future CDS developers.

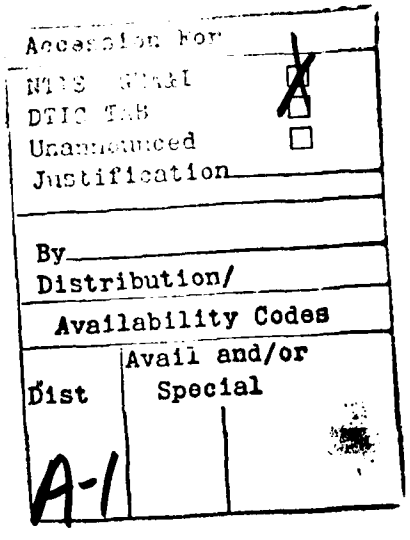

TABLE OF CONTENTS

I. INTRODUCTION

This thesis addresses the need for portable user interfaces for real-time Ada command and control programs. It is a part of ongoing research for developing a Low Cost Combat Direction System (LCCDS) at the Naval Post Graduate School.

A. BACKGROUND

1. Combat Direction System (CDS)

Development of the LCCDS is sponsored by the Naval Sea System Command (NAVSEA) as a cost effective approach to developing CDS capabilities for non-combat ships. LCCDS will require a platform independent windowing system supporting graphical capabilities and implemented on commercially available microprocessor-based workstations. The goal of the system is to enhance the operating capability and decision support resources of commanders of naval non-combatant ships[Ref.1].

2. Ada Programming Language

The United States Department of Defense (DoD) sponsored the development of a single high order language[Ref. 3] for development of a large embedded system such as CDS—Ada. The Ada language provides portability, reusability, maintainability, reliability, and real-time embedded computer system capabilities. However, Ada does not directly possess capabilities to deal with the windowing environment also required by CDS.

3. X Window System

The X Window System (or simply X) is the only system currently available which meets the platform independent requirement of CDS. X is a network-transparent window system. Multiple applications can be run simultaneously in windows, generating text and graphics in monochrome or color on a bitmap display. Every window can be thought of as a "virtual screen" and can contain subwindows, to an arbitrary depth. Windows can overlap like papers on a desk and can be moved, resized, and reordered dynamically. Such a system provides the equivalent of many terminals and provides integrated access and manipulation of data for command and control (C^2).

4. User Interface

A CDS is a type of decision support system (DSS). A DSS may be defined as an interactive computer system that supports the cognitive processes of judgement and choice. Such systems are not primarily concerned with mere storage and retrieval of data, but with combining a model base, data base and dialog such that decision making processes are optimized. The CDS data base contains target, intelligence, and environment information. The dialog or user interface provides the interactive screens, dialog boxes, menus, graphic displays etc. The dialog also implements the decision model and gives the user an optimal view of the current tactical environment.

B. STATEMENT OF THE PROBLEM

In C^2 environment, the CDS manipulates and presents real-time combat information to a commander as efficiently and effectively as possible to optimize the decision making process. A LCCDS can be developed using off-the-shelf hardware and available windowing software. However, the software interface to X is written in the C programming language rather than Ada. Standard methods for interfacing Ada based systems to X are needed. Additionally, different Ada programs require different types of user interface. This thesis lays ground work for future development of user interfaces for C^2 systems like CDS by describing several mehtods for linking Ada real-time programs to an X based user interface.

C. RESEARCH APPROACH

1. Background Study

This research began with a study of user interface requirements for CDS, along with current software technology related to Ada and the X windows environment. A thorough review of the literature was accomplished to provide understanding of this information. A summary of that review is described in chapter 2.

2. Define and Explore Ada—X Windows InterfaceAlternatives

Three user interface categories are defined including I/O models, language interface requirements and program/user interface control models.

3. Experiments : Description and Results

Example programs are described and results and displays of tests are discussed for each alternative interface method. All experiments are based on a small, real-time, program modeling a standard digital stop watch.

II. RESEARCH STUDY

A. COMBAT DIRECTION SYSTEM

Combat Direction Systems support the processes of command and control and provide communications and intelligence for battle force operations. Commands or decisions result in feedback to an environment—control. Decision-making is supported by both communications and intelligence, while control is supported by communications[Ref. 6]. One of the primary roles of Battle Force Combat System support is information exchange, synthesis, and filtering. The Battle Force Combat System is not any single procurable system, but is, rather, a collection of systems that support Battle Force command. CDS is one such system.

1. Functional Description

CDS provides command, control and communication (C^3) capabilities at the ship level. It is the composite of shipboard elements and personnel that includes detection, tracking, identification, processing, evaluation, and control of engagement of hostile threats, either actively or passively. The CDS may include other subfunctions such as navigation, testing, and training. A diagram of the AEGIS CDS is show in Figure 2.1.

A CDS may be defined as those combinations of men and data handling systems, either manual or automated, employed to execute combat direction functions. They support command at levels from the platform (ships, aircraft, submarines) up to, and including the task group or force.[Ref. 4]

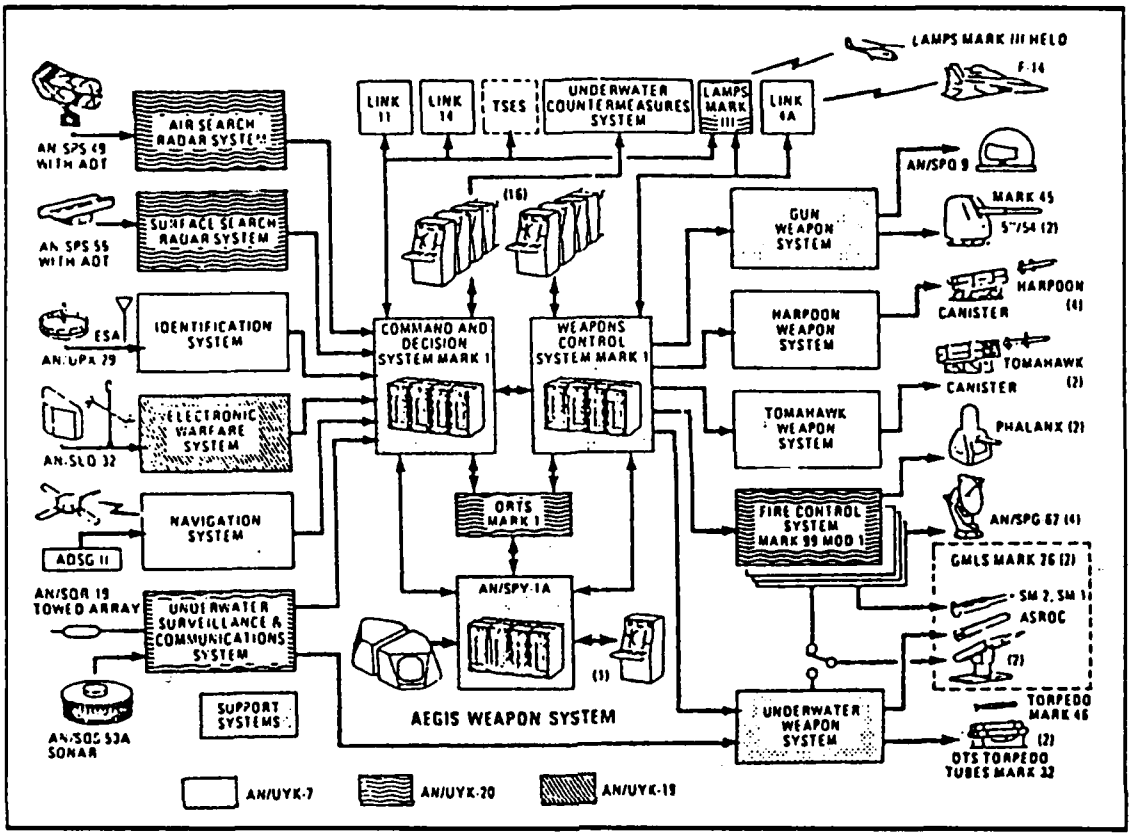

Figure 2.1 Combat System diagram[Ref. 5]

An automated CDS consists of a complex unit of data inputs, user consoles, converters, adapters, and radio terminals interconnected with high speed general purpose computers and their stored programs. By providing information about the overall tactical situation, the CDS enables the force commander to make rapid, accurate evaluations and decisions.

Current systems have progressed to a level of sophistication that includes upwards of 20 tracking interfaces including multiple tracking sensors, multiple weapons interfaces, electronic warfare and multiple tracking data link systems[Ref. 1].

2. NON-Combat Version

NAVSEA proposes to implement a version of the CDS on naval non-combatant ships. The use of commercially available microprocessor workstations and software represents a low cost approach to providing a sophisticated capability on board every naval ship that leaves the inner harbor, and potentially enhances the over all power of the fleet. General requirements for CDS on Non-combat ships are as follows[Ref. 1] :

- The system must be able to rapidly summarize and display the information needed by the operator, provide functions for manual or automatic tracking and identification (ID), allow the CDS operator to build and display a set of geographically stable and/or moving point of information such as threat/non-threat ships, aircraft, shoreline maps.
- The system must be flexible to adapt to various sensors and communication media and include manual input. It must also be able to adapt to changing threats and environments.
- The software must be portable to allow distribution to different computer platforms, and enable incorporation of advances in hardware[Ref. 6].

3. Hardware Requirements

The system hardware is a cooperative effort between Navy and industry to meet Next Generation Computer Resource (NGCR) requirements. The NGCR specifies a set of state of the art computers for shipboard use in the future with the follows features[Ref. 1]:

- 32 bit ISA(Instruction Set Architecture)
- 1 - 100 MIPS(Million Instructions Per Second) performance depending on application requirement
- 26,000 hour Mean Time Between Failure
- One or more of the industry and government back plane bus standards including VME, IEEE 896 (future bus), IEEE 1296 (Multibus II),and VHSIC PI-BUS
- Three types of local area networks to choose from depending on the application requirement . These include:
    - SAFENET I (<10 Mbps)
    - SAFENET II (>100 Mbps)
    - High Performance LAN (~1 Gbps)
- Network Data Base Management System

Since ruggedized Sun workstations which meet this NGCR are readily available and are used in both US Army and US Air Force C^3 systems, this study will use compatible Sun3/Sun4 workstations. Work accomplished on these systems should be easily portable to similar hardware from other vendors.

4. Software Requirements

DoD Directive 3405. 1, April 1987 requires the use, in priority order, of the following[Ref. 1]:

- Off the shelf applications.
- Ada based software and tools.
- Other approved high order language.

Given this requirement, any CDS software component must be either one of the above or a combination of the above. The NPS prototype CDS effort, of which this study is a part, will primarily be an Ada based software development integrating off-the-shelf software packages when available[Ref. 1].

The selected environment for the LCCDS system will be a computer system with a standard operating system (UNIX) and software environment that supports CDS. This software environment must have a multiprocessing capability. That is each application will be considered a separate process with its own internal private memory and may ideally run on its own processor/hardware. Applications will communicate through messages adhering to a specific protocol. This protocol establishes the visible interface of each application.

Object-oriented approaches for designing graphical user interfaces have been shown to be effective, especially when combined with facilities for managing constraints[Ref. 7]. This approach is proposed for constructing the user interfaces of the LCCDS and will be discussed in more depth in section C.

B. ADA PROGRAMMING LANGUAGE

1. Real-time Ada and CDS

Ada is a high level programming language with considerable expressive power. It was developed under the auspices of DOD specifically for use in large real-time embedded computer systems[Ref. 3] such as CDS. The CDS will require real-time control of many concurrent processes, automatic error recovery, fail-safe execution, and interfaces to many complex nonstandard input/output devices. Ada also sup-

ports the use of modern software development principles, including top-down structured design, modularity, data abstraction, and information hiding. Ada offers promise for improvements in the critical areas of software maintainability, reliability, and reusability[Ref. 8].

2. Portability and Reusability

Use of Ada will promote the movement of software from one type of computer to another—both because the language has been standardized and because implementation dependencies can be localized or isolated. This facilitates both the sharing of Ada software among systems using different hardware platforms (reusability) and the replacement or upgrade of computers within a system without major software modification[Ref. 9].

Ada allows the software to closely model the environment in which a set of objects interact with one another via a limited set of operations. Ada packages are used to encapsulate system objects with the operations which manipulate these objects.

3. Maintainability and Reliability

The Ada language encourages a disciplined programming style with uniformity in format among program modules. Even though an embedded computer system might contain hundreds of program modules, Ada's packaging, modeling as abstract state machines, data types, and concurrent processes makes it unnecessary for the programmer to comprehend all levels of detail in order to acquire a basic understanding of a software architecture. Since Ada is a strongly typed language, extensive type

and range checking occurs at both compile and real time. This technique greatly improves software, reliability[Ref. 10].

4. Real-time Capabilities

Ada provides unique features for handling real-time concurrent processing not found in other languages. Tasks are the Ada program units which define operations which many execute in parallel. Ada provides its own run-time system to handle synchronization and communication between tasks. Such communication and synchronization is called a rendezvous[Ref. 6]. This interaction allows one task to select and report the interaction or improper action of another task enhancing communications reliability and error detection[Ref. 11].

Since in the real-time problem space, tasks operate in a highly parallel fashion (more than one event occurring at the same time), the task program unit serves to greatly reduce the distance between the problem space and the solution space by eliminating the need to convert real-time parallel events into the serial abstraction demanded by other high-order languages[Ref. 12].

5. Benefits of Using Ada

A summary of the benefits of Ada follow[Ref. 9]:

- The Ada language is required by DOD as the standard programming language in mission critical computer systems and is the NAVSEA required language for CDS software design[Ref. 1].
- The Ada language can be used and ported to various on commercial computers for which commercially developed Ada compilers are available.

- The Ada language is expected to improve the quality of software and reduce the cost of software development and maintenance.
- The Ada language was designed to support real-time requirements of embedded computer systems.
- The Ada language was designed to support teams of people who are jointly developing large, complex software systems such as CDS.

C. THE X WINDOW SYSTEM

1. Workstations with the X Window System

Workstations integrate computer graphics, bitmapped video display, keyboard entry, and pointing device entry into a single package. Multi-window capabilities simplify multiple process control since the user can visualize several different concurrent jobs simultaneously[Ref. 13]. A single workstation often takes the place of several individual computer terminals.

However, developers of workstation application software are faced with the problem and expense of developing a different graphical interface for each type of workstation hardware. Because developing workstation applications requires reprogramming each application for each workstation, a standard graphical interface is desired to reduce effort and cost. The X window System offers a solution to this problem by providing a common and portable interface to workstation hardware[Ref. 13].

2. X Windows System Environment

The X windows operating environment supports multiple overlapping windows on a variety of color and monochrome workstations. Network transparency means that X applications many run on one cpu and display input/out through another cpu using a display at the latter location. It allows a variety of computing styles ranging from stand-alone workstations running applications locally, to time-shared mainframes using workstations as if they were terminals. All these styles can coexist in the X environment[Ref. 14] as shown in Figure 2.2.

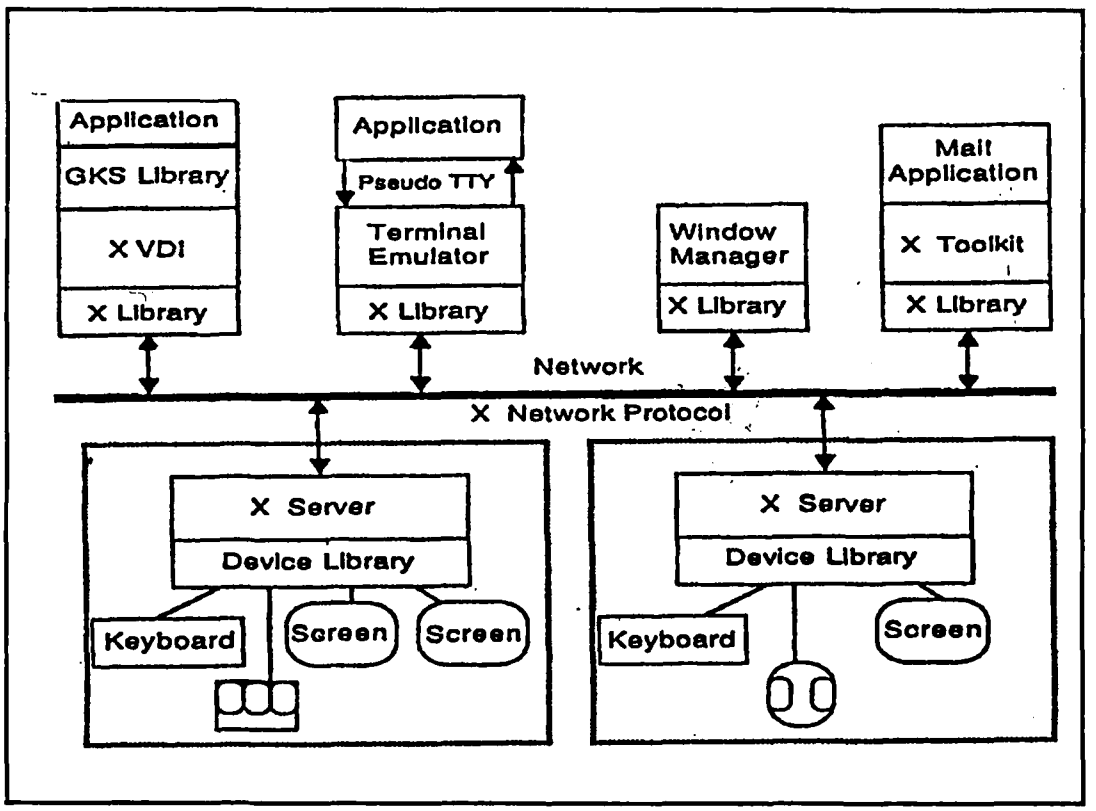

Figure 2.2 X Window System Structure[Ref. 17]

X workstations can run many different applications on any cpu in the network. Each of these applications can use as many windows as it needs to display output, and can receive input. Likewise, a single X application program can display output windows and accept input on many different workstations at the same time.

3. X Windows System Environment

The foundation of X Windows is the base window system. The base window system interfaces with the outside world using the X network protocol. The network protocol interface is designed to work either within a single central processing unit (cpu) or between cpus, making it excellent for multiprocess or distributed applications such as embedded systems[Ref. 14].

Rather than use the X network protocol directly applications work through a programming interface called Xlib (see Figure 2.3). Application programs many alternatively use high-level X toolkits to mask the complexity of Xlib. All user interfacing need not be programmed explicitly, but may be provided by an X window manager. Window' managers are special applications for managing user workstation. The user work space can be managed by special applications called managers. They allow users to activate, deactivate and control I/O to concurrent applications; to resize, move and iconify windows.

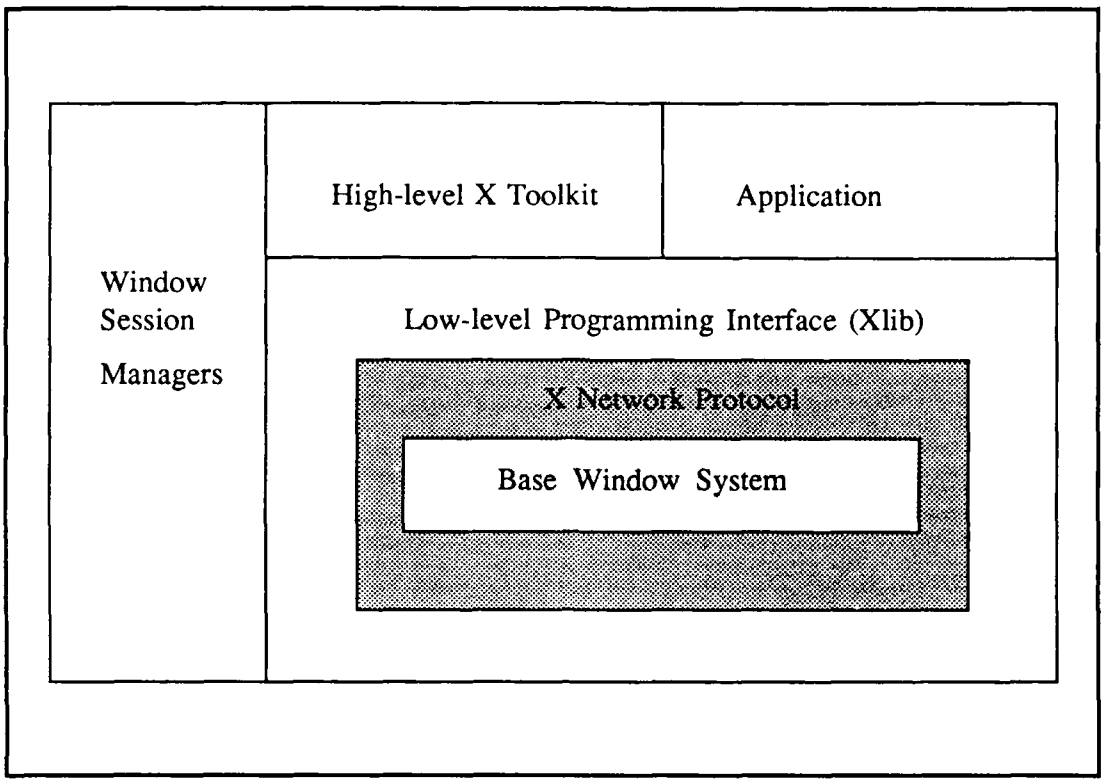

Figure 2.3 X Window System Workstation[Ref.14]

4. X Window User I/O handling

X applications handle user input as follows[Ref. 13] :

- Users generate input by pressing keys on the workstation's keyboard or manipulating the workstation's mouse/pointing device.
- The window manager captures events and processes on to application.
- The base window system distributes this input to applications in the form of events.
- X supports a large variety of typewriter keyboards.

X's base window system offers a full-featured suite of bitmapped graphics operations. Window managers, applications, and high-level toolkits use X's graphics operations to draw information on the workstation's. Output capabilities follow [Ref. 16]:

- X organizes display screens into a hierarchy of overlapping windows. Each application can use as many windows as it needs, updating resizing, moving, and stacking them on top of one another as needed.
- X provides drawing capabilities. X's graphics operations are immediate rather than display-list-oriented: the workstation does not save a series of graphics operations, but rather draws everything immediately.
- X lets applications draw high-quality text in variety of fonts. X's text supports applications, ranging from video-display-terminal emulators to multi-lingual word processing programs.
- X's drawing operations are relative: applications specify all operations in terms of integer pixel addresses within windows, so applications can draw things in their windows without regard to where their windows are positioned on the screen.
- X can output in either color of monochrome (black-and-white).
- X can display, manipulate, and capturing bitmapped images.

III. ALTERNATIVE ADA-USER INTERFACE METHODS

A. GOAL

The goal of this research is to find methods of interfacing Ada programs with X based user interfaces. Currently, interfaces to X are only available in C and C++ through Xlib and high level toolkits. To use these toolkits, Ada must be able to interface to C/C++. The methods described here will provide future CDS user interface designers a step-by-step means to select and design the user interface for CDS.

B. TOOLS

Several tools were used for researching X user interface development. These include X facilities, related languages and X toolkits. The following sections are to explain each of these areas.

1. X Facilities

X facilities a user interface may use include X resources, window managers, and the Xlib library. X itself provides an object-oriented architecture where X classes can have resources tailored by the user. X applications create, use, and destroy resources in the course of their operation. Resources are essential to the way applications interact with workstations. They allow applications to transmit state information, such as window positions and drawing colors to the workstation just once, then use them repeatedly. Applications manipulate the following types of resources: win-

dows, graphic contexts, fonts, color maps, pixmaps and cursors. The purpose of resources is to allow applications to control the state of the workstation explicitly [Ref. 2].

Users can invoke multiple application windows within an application by using an X window manager. Applications contain both transient windows and permanent windows, but only the permanent windows are manipulated by the window manager. A user can invoke a window manager to change the size, position, or state of any top-level window. By convention, a window manager can put managed windows into one of five user visible states: Normal, Iconic, ClientIcon, Inactive, and Ignored, and can change the state of any window upon user request. The Normal state indicates normal operation; Iconic is when the window is unmapped and represented by an icon; ClientIcon means the application wants its icon window to be visible, Inactive is represented by an application Menu rather than an icon; and, finally, Ignored where the window is ignored by the window manager[Ref. 2].

Xlib is the subroutine library used within an application program to interact with an X workstation server. When the application calls an Xlib function, it generates X protocol requests and sends them to the workstation to be handled by the X server. Xlib is the lowest level application programming tool for X and as such, was not used for this research. Instead, a high level toolkit based on Xlib was used and is described.

2. Languages

Three programming languages were used in this research: Ada, C, and C++. Ada was required for applications as described in Chapter II and is used exclusively for implementing CDS functionality. C++ is used for X based user interface development and C is used for simple UNIX system utilities and drivers.

C is often referred to as a low level/high level language because it provides tools that allow low-level operations to be performed while retaining the advantage of a high level block structure. C provides direct access to UNIX system functions and is very powerful and efficient for such uses. However, C's cryptic readability and lack of strong typing make it undesirable for programming large real-time systems.

C++ is based on, and is a superset of C. C++ retains C's power and flexibility to deal with the hardware-software interface and low-level system programming while providing a platform to support object-oriented programming and high-level problem abstraction. C++ allows the set of attributes of an abstract object and the associated set of operations or functions to be encapsulated in a class definition. The class provides the basic underpinning for object-oriented programming and serves as the vehicle for data abstraction and data hiding. C++ allows subclasses to be defined and provides a natural means of extending the language. C++'s object-orientation makes it the language of choice for graphic user interface toolkits.

3. X Toolkits

Using X Windows toolkits and their facilities, such as scroll bars, menus, dialog boxes and icons, supports data hiding and protection, extensibility and code shar-

ing through inheritance. Toolkits reduce user learning time, limit the rate of error by the user and enhance user performance. The user interface toolkit chosen for this research is InterViews[Ref. 15].

InterViews provides a library of predefined objects and supports the composition of a graphical user interface from a set of interactive objects. A user interface is created by composing simple primitives in a hierarchical fashion, allowing complex user interfaces to be implemented easily. InterViews supports the composition of interactive objects, such as scroll bars and menus, and graphics objects, such as circles and polygons. InterViews was developed at Stanford University and is one of the best user interface environment toolkits available today. InterViews was chosen for this research for the following strengths[Ref. 15]:

- InterViews provides a simple organization of graphical interface classes that is easy to use and extend via subclasses. Subclasses such as box, tray, deck and frame make it possible to compose interactive components into complete interfaces without specifying layout details. Figure 3.1 shows how objects from the InterViews library are incorporated into an application.
- Abstract and interactive behavior are separated into subject and view objects to support different interfaces to the same functionality. For example, a shoreline map might be stored as a object in a data base. A view of that object may be manipulated by moving and zooming the window, or simultaneous views may be manipulated without modifying the underlying subject. Conversely, a change to the subject will automatically update each of the current views.
- The InterViews library completely hides the underlying window system from application programs. This means that InterViews applications can be transferred to a new window system simply by porting the primitive classes. Figure 3.2 shows the relationship between levels of software supporting the application.

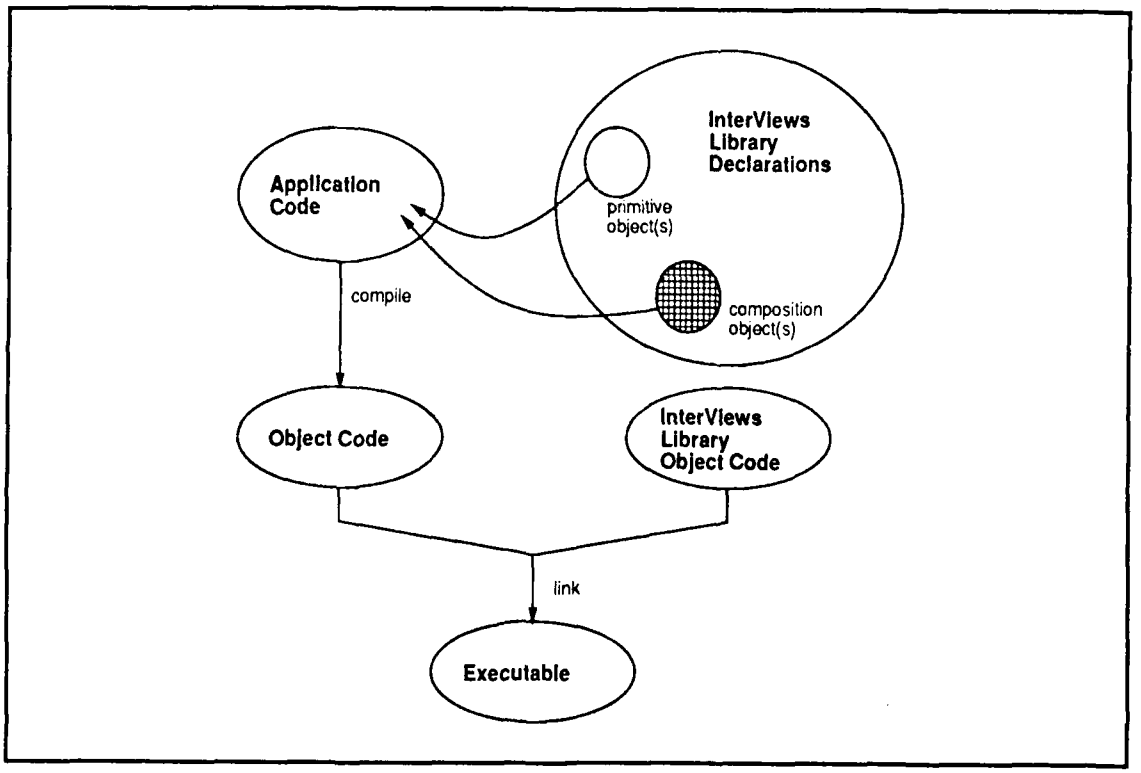

Figure 3.1 Incorporating InterViews objects into an application[Ref. 15]

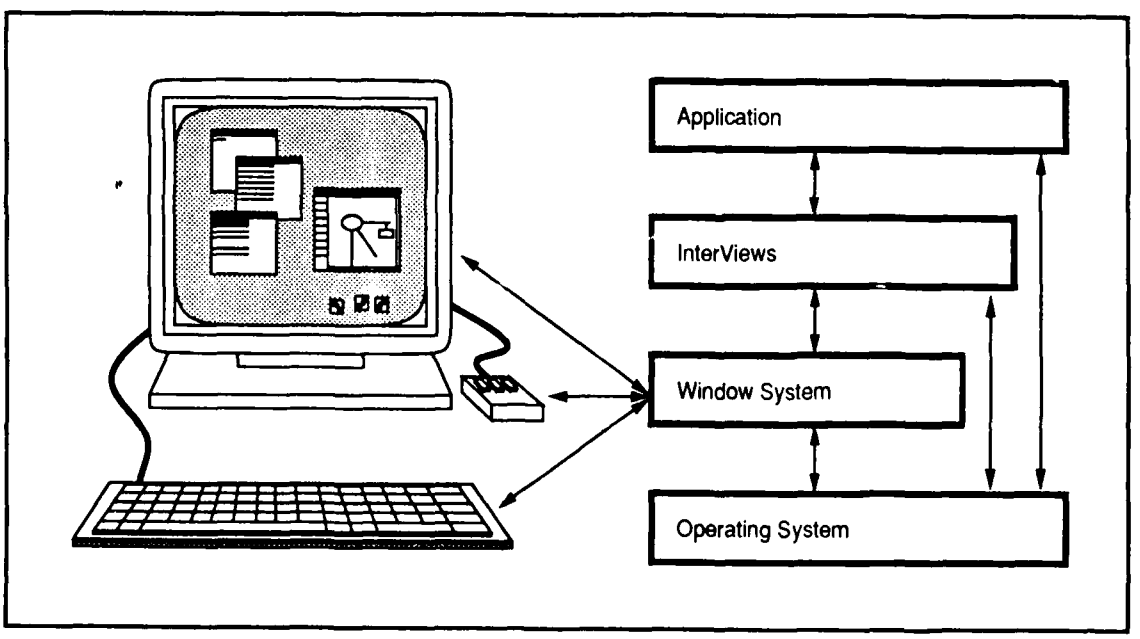

Figure 3.2 Layer of software underlying an application[Ref. 15]

- Using an object-oriented language (C++) to implement InterViews provides a package that is both simple to use and easy to extend. InterViews object-oriented terminology and methodology provide a natural way of expressing interactive behavior.

C. USER INTERFACE ALTERNATIVES

Analysis of user interface requirements for CDS [Ref. 17] indicates the following issues need to be addressed: the user input/output model, the programming language interface required, and the appropriate control model. The I/O model refers to the means of user interaction such as line-oriented, cursor-oriented, or window-oriented. The programming language interface is based on toolkit or system/library usage and may require interfacing Ada with C or C++. The control model deals with whether the application or the user interface is in control, or whether both run as communicating asynchronous processes.

1. User Interface I/O Model

a. *Line-oriented*

An application with a line-oriented user interface uses text command line interaction from either a shell or console window. The application usually displays a list of choices or a prompt, and the user answers by typing a numbered selection or by answering the question. Such user interfaces are common on character based terminals with only a teletype mode of communication.

b. *Cursor-oriented*

A cursor-oriented interface consists of interaction through the use of a standard terminal or terminal emulator such as an Xterm. In this style, applications directly manipulate the cursor within a text-oriented window to provide a form based dialog. Each screen is formatted with various prompts and blanks the user fills in. New screens are automatically generated as appropriate responses to user selections and inputs. The cursor is manipulated via the keyboard arrow keys or a mouse and moves from selection to selection rather than randomly to independent points on the screen.

This type of user interface is provided in UNIX systems using the CURSES library. CURSES is a terminal independent package of display functions, data types and objects provided in the system library. The CURSES library can be called directly by C programs and many Ada compilers contain a package binding to it. Basic functions found in CURSES include those that [Ref. 16]:

- move the cursor to any point on the screen,
- insert text anywhere on the screen, normal or highlight mode,
- divide the screen into rectangular areas called windows,
- manage each window independently,
- draw a box around a window using any character.

c. *Window-oriented*

Windows provide many advantages that normal terminals can't provide. They make it easy for applications to organize the workstation screen into areas for particular purposes, such as text display, graphical display, and menus. Windows

allow multi-window separate processes to be executed and, permit windows to overlap. Users can move, change size, scroll or close windows. Applications can be executed by selecting the appropriate Icon or by opening a menu and using the mouse to select the desired item[Ref. 2].

2. Language Interface

a. *Ada Only*

For simple line-oriented or some cursor-oriented applications, no interface with another language is required. Ada can be used to implement the input/output requirements using available text, or graphics packages.

b. *Ada—C*

When applications need to access system dependent functions, a C interface may be required. Specific types of interaction include Ada defining C data structures, Ada calling C subprograms or accessing C objects, and C calling Ada subprograms. Binding to such implementation dependent features falls under the auspices of the Ada Language Reference Manual Chapter 13 [Ref. 3]. The Verdix Ada Development System (VADS) [Ref. 18] uses several pragmas and storage definition techniques for defining these implementation dependencies. Applicable Ada—C interface schemes are described in the following paragraphs.

(1) Ada defining C data structures

When data must be passed or shared among Ada and C subprograms, a common data structure must be defined. This data structure must have exactly the same binary implementation in both languages. Two basic approaches are available for creating parallel data types: using a priori knowledge and using Ada representation specifications. There are some types that the programmer knows are parallel between two language implementations from reading the vendor's documentation. Some samples of the more common C simple types and their corresponding Ada predefined types are given below.

C	Ada
int	integer
float	short_float
short	short_integer
char	character

VADS provides Ada with the package C_TO_A_TYPE which defines parallel data types that can be used for a C language interface. The Ada int and char types from C_TO_A_TYPE are shown below.

```
type int is range -(2**31)..(2**31)-1:
  for int'size use 32;
  -- VADS implementations with 32-bits integers
type char is range -(2**7)..(2**7)-1;
  for char'size use 8;
```

Record types (C structures) are more complex. A slightly modified example from the VADS manual [Ref. 18] below shows the C structure, the corresponding Ada record, and the necessary bit level storage allocation scheme.

```
/* C definition of the TTY data structure from CURSES */

struct ttyb{
  char ispeed;
  char ospeed;
  char erase;
  char kill;
  short flags;
};
```

```
-- Ada definition of the TTY data structure

type C_SHORT is range -(2**15)..(2**15) -1;
type C_TINY is range -(2**7)..(2**7) -1;

type TTYB is
  record
    ISPEED : C_TINY;
    OSPEED : C_TINY;
    ERASE  : C_TINY;
    KILL   : C_TINY;
    FLAGS  : C_SHORT;
  end record;

for TTYB use
  record
    ISPEED at 0 range 0..7;
    OSPEED at 1 range 0..7;
    ERASE  at 2 range 0..7;
    KILL   at 3 range 0..7;
    FLAGS  at 4 range 0..15;
  end record;
```

Again, a priori knowledge can be used. Both C and VADS associate the record label with a base address from which offset to access individual components of the record are calculated. As long as the record is composed of equivalent simple data types, the offsets will be calculated similarly and the record structure will be identical.

(2) Ada calling C subprograms and accessing C objects

To let an Ada program call a C subprogram, the subprogram name must be made global and identified to the Ada program. This is accomplished first by declaring the Ada subprogram, and then by using the predefined pragma INTERFACE to establish a link from the Ada procedure or function name to the corresponding C function or macro (the link name is formed by prepending an underscore '_' character to the C program symbolic name). Pragma INTERFACE uses the format shown below:

```
procedure ADA_PROGRAM_NAME;

pragma INTERFACE (C, ADA_PROGRAM_NAME,
  "_C_program_name");
```

Pragma INTERFACE_OBJECT allows access to external variables. The first argument to the pragma is the name of an Ada variable; the second is a string that gives the linker name of the external variable. As with subprograms, the object name must be declared before the pragma as an object of an Ada type compatible with a C type. For example:

```
Ada_Variable : Ada_C_Compatible_Type;
pragma INTERFACE_OBJECT (Ada_Variable, "_C_variable");
```

(3) C calling Ada subprograms

To let a C program call a Ada subprogram, the C program name must be made global and identified to the Ada subprogram. The pragma EXTERNAL_NAME allows a specific link name to be given to an Ada subprogram so that

it might be called from another language, Pragma EXTERNAL_NAME uses the format shown below:

```
-- Ada subprogram called by C
procedure ADA_PROGRAM_NAME;
pragma EXTERNAL_NAME (ADA_PROGRAM_NAME,
  "_C_program_name");

procedure ADA_PROGRAM_NAME is
begin
--operation;
end ADA_PROGRAM_NAME;

/* C code calling Ada program */

extern C_program_name();
main(){
   C_program_name();
}
```

In this example, the internal symbolic name is the string '_C_program_name.' The link name is formed using the same rules as for the pragmas INTERFACE and INTERFACE_OBJECT.

c. *Ada—C++*

Ada can be linked with C++ InterViews programs using the same techniques defined previously for C programs. However, care must be made to get correct linker names for C++ objects and methods. Exact linker names can be obtained using the UNIX ***nm*** command. For example:

```
// C++ test program: test.c

#include <stream.h>
class Test{
    static i;
  public:
    Test();
    ~Test();
    void look_around();
};

#include "test.h"
Test::Test(){i++;}
Test::~Test(){i--;}
void Test::look_around(){
 if(i>1)
  cout<<"there are other objects of this class\n";
 else
  cout<<"there are no other objects of this class\n";
}
// after compile, nm test.o results:

00000004 C _Test$i
00000024 T __$_Test
00000010 T ___4Test
000000e4 d ___vtbl__circbuf
000000fc d ___vtbl__filebuf
00000114 d ___vtbl__streambuf
000000a0 T _look_around__4Test
00000000 t gcc_compiled.
```

Note the linker names for the class test methods: __$_Test, ___4Test, and _look_around__4Test, and the static variable i : _Test$i. These are the names which must be used in the Ada EXTERNAL_NAME and INTERFACE_OBJECT pragmas.

3. Application—User Interface Control Model

An important issue to clarify in any user interface is who is in charge: the application program or the user interface program, or whether both are running as separate processes. In other words, does the application control the user interface ob-

jects and behavior, or does the user interface control execution of application subprograms, or does the user interface and application run asynchronously and communicate using global data or message passing. These alternatives are illustrated in Figures 3.3 and 3.4. The following sections discuss each condition. The mail box method requires strict control to provide data integrity. Data integrity is compromised when one process is faster than the other process. Data can be lost if the consumer is slow, or resources wasted if the producer is slow. Semaphores or busy/wait flags can be used to solve the problem, but care must be taken to avoid deadlock.

a. *Ada application-controlled model*

Figure 3.3 shows an application program using user interface objects to output and receive data from the user. In this case, the Ada program is blocked while the user interface subprogram is processing. The implementation of the user interface subprogram is hidden from the application. The user's view may be changed without modification of application functionality. This type control model causes disjoint dialogs and can't maintain single persistent views due to the blocking of the application.

b. *User interface-controlled model*

Figure 3.4 shows a software structure in which Ada application subprograms are independently invoked by the user interface interactive objects. Again access to the user interface toolkit is hidden from the application. The user interface runs as a continuous function, processing user initiated events and calling application subprograms to execute major functions. While the interface is consistent, it is blocked while applications functions are being executed. Complex applications conse-

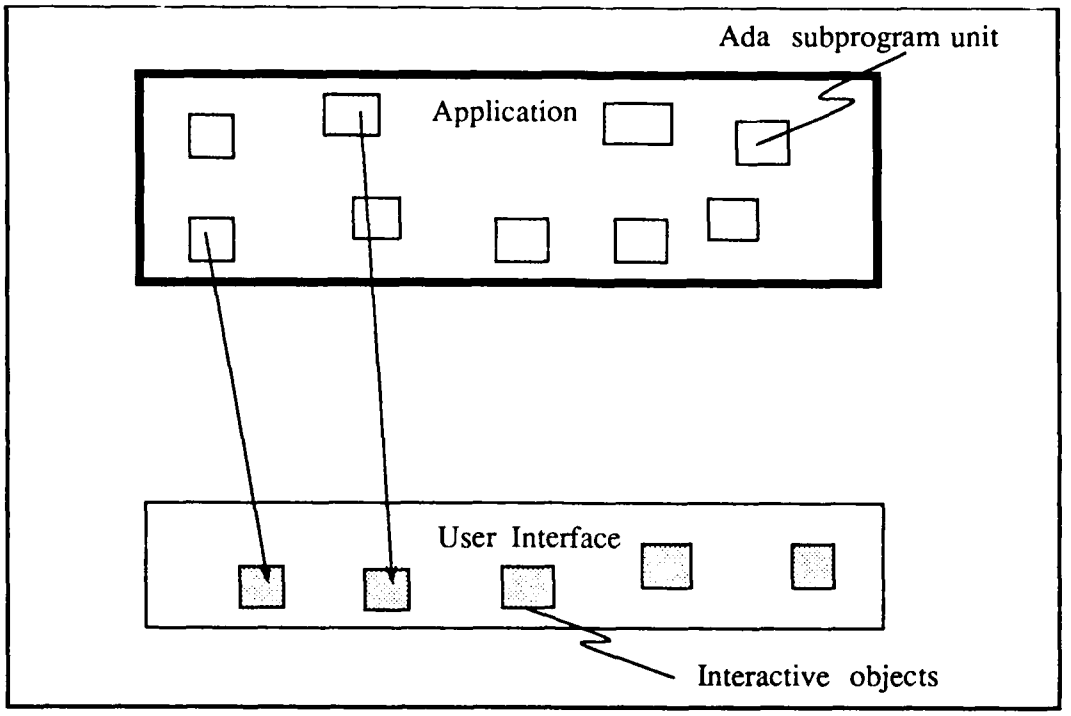

Figure 3.3 Ada CDS program control of the user interface

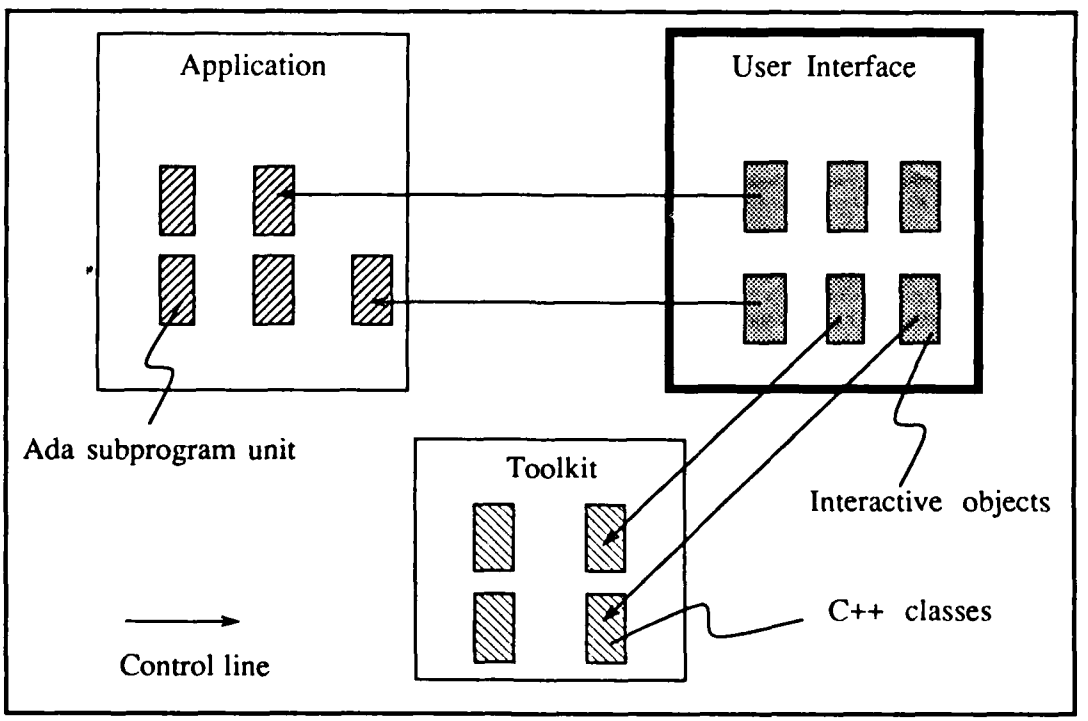

Figure 3.4 User Interface control of the Ada program and toolkits

quently slow down response time. This technique is not well suited to a real-time application since the Ada programs are activated only under user interface control rather than external sensors. In addition, the application software architecture is reduced to disjoint small functions, negating the structural advantages of Ada packages and tasks.

c. *Asynchronous-controlled*

Two problems must be solved related to asynchronous control: how one process identifies or initiates the other, and how processes communicate. Typically one program starts the other as a child process. Processes communicate using either message passing or mail boxes (shared memory buffers). Since user-interface—application communication is in the form of graphic or large text data, the mail-box or buffer is more efficient than message passing. The mail box is a global data structure visible to both processes but allocated as a separate resource by the operating system. The UNIX system commands needed for asynchronous operation are listed below[Ref. 19]:

- shmget—get shared memory segment,
- shmat—attach shared memory to a process,
- fork—create a new process,
- execlp—execute the new process,
- shmctl—shared memory control operations

To use system commands, an interface to the system command must be established similar to that described previously for calling C programs. A simpler method, and one which allows the combining system commands into a single specific func-

tion involves the initialization of shared memory for inter-process communication, involves writing a unique C utility program. This C program will define a single C function which will be called by the application to fork the user interface process. A sample program is shown below:

```
/* spawn the user interface */
spawn_UI(){
   if (fork() == 0){
      execlp("UI", "UI", null);
      perror ("fork of user interface failed");
      exit(-1);
   }
 }
```

Thus rather than binding to all required system functions, bind only to one C function with no parameters. Similarly, C programs can be written to create/destroy the shared memory needed to implement the mail box scheme. Shown below are C structures and subprogram examples for establishing shared memory communication.

```
/* shared memory data structure */

struct data {
char data;
};
struct data *shm_pointer;

/* address for the memory */

key_tmemory = 123456L;

/* get memory ID          */

intmid;
int shmflag = IPC_CREAT|IPC_EXCL|0666;
```

```
void open_share_memory(){
   int i;

/*create a shared memry segment for the user interface*/
   mid = shmget(memory,sizeof(struct data),shmflag);
   if (mid < 0) {
       mid = shmget(memory,sizeof(struct data),0200);
   }

   /* attach our process to it */
   p = (struct data *) shmat(mid,(char *) 0, 0);
   if (p == (struct data *) -1)
       perror("server:shmat");
 }

/* remove the shared memory segment */
void close_share_memory(){
     mid = shmget(memory,sizeof(struct data),0200);
     if (shmctl(mid,IPC_RMID) == -1)
        perror("server:shmctl(remove)");
        exit(0);
}
```

4. Summary

Table 3.1 shows the combinations of I/O model, language interface, and control models considered. The line-oriented I/O model is easy to perform with Ada's TEXT_IO so no language interface is required. Asynchronous I/O can be implemented using Ada tasking. Cursor-oriented I/O may require interfacing to C if no cursor-control package is available. With this model, asynchronous processes may still be handled within Ada using tasking as long as an Ada cursor package exists or is written.

User interfaces using the window-oriented I/O model may require linking the Ada/C++ programs if synchronized control is all that's needed. The asynchronous model, however, requires the additional steps of initializing shared memory and forking the user interface process.

Table 3.1 Summary of user interface methods

I/O MODEL	LANGUAGE INTERFACE	CONTROL MODEL
Line-Oriented	None	Application-controlled
		Asynchronous
Curses-Oriented	Ada—C	Application-controlled
		Asynchronous
Window-Oriented	Ada—C++	Application-controlled
		User interface-controlled
	Ada—C C++—C	Asynchronous

IV. EXPERIMENTATION

The example used to illustrate the user interface design methods discussed in the previous chapter is a simple digital stop watch. The stop watch contains multiple concurrent processes handling real time events. Methods of user interface design are shown by modeling the stop watch using various combinations of user I/O model, control model and language interface. These variations are presented and discussed in the following order:

- Experiment 1: Line-oriented user interface using TEXT_IO
- Experiment 2: Cursor-oriented user interface using Ada CURSES
- Experiment 3: Cursor-oriented user interface using C CURSES library
- Experiment 4: Window-oriented, application-controlled user interface using InterViews
- Experiment 5: Window-oriented, user interface-controlled application using InterViews
- Experiment 6: Window-oriented, asynchronous-controlled user interface using InterViews
- Experiment 7: Execution of stop watch by window manager menu
- Experiment 8: Execution of stop watch by a command line
- Experiment 9: Execution of stop watch by an icon

A. STOP WATCH PROGRAM (SW)

The SW program is a representation of a simple digital chronograph and provides start, stop, lap, and reset functions. Elapsed time, lap time, and number of laps are displayed. All times are in 1/100 second and laps range from 0 to 9.

This program is developed using three packages. The dependency diagram showing the program architecture is shown in Figure A.2 in Appendix A. The chronograph package contains a timer task and is a state machine used to represent the current state of the stop watch (initial, running, stopped). As such, it also contains the control functions needed to change the SW state and update the elapsed time, lap time, and current lap. The SW operations are implemented as timed task entry points and the timer values are updated at an interval specified by the user interface.

The task entries include the four basic commands: *start, stop, lap* and *reset*, as well as an additional *kill* function. The function *start* changes the timer state to *running* and requests an immediate update of the display time. The function *stop* changes the state to *stopped* and also calls for an immediate update of the timer. If *lap* is invoked when the timer is *stopped*, previously recorded lap times are displayed. When the state is *running* and *lap* is called, it records the current time associated with a lap number. The *reset* function initializes all timer and lap values to zero, sets the state to *initial* and causes display of both lap and elapsed times.

The display package represents the user CRT and keyboard and provides get and put operations to the chronograph. Both operations are in terms of chronograph data types. These operations translate chronograph data types into data types understandable by the user interface. In most cases, these are text strings or ASCII characters. Display also provides the update interval to the chronograph since the update interval is dependent on user interface type (for example, a line-oriented user interface defines a larger interval to allow the user time to input commands).

The user interface package provides the actual user interface implementation based on the discussion of the previous chapter. In most cases, only the user interface body need be changed to implement a new user interface style. Basic user interface operations are OPEN_UI, CLOSE_UI, GET_COMMAND, UPDATE_TIME, and UPDATE_LAP_TIME. In general, OPEN_UI initializes display data and the user interface window. CLOSE_UI terminates the user interface gracefully when required. UPDATE_TIME provides the time to displayed to the user interface, and UPDATE_LAP_TIME provides the lap time and number to the user interface. The implementation of these operations is dependent on the type of user interface. This software architecture does not represent the most efficient implementation, but is used to illustrate the principles of user interface independence, abstraction, and information hiding. The code for the basic stop watch program is given in Appendix A.

B. LINE-ORIENTED USER INTERFACE

In the line-oriented user interface, the application uses text command line interaction from either a shell or console window. Processing alternates between the user typing commands and the application giving feedback. Commands are implemented as single characters: 's': start, 'q': stop, 'l': lap, 'r': reset, and 'k': terminate. Feedback consists of one or two lines giving the elapsed time and/or lap number and time.

Experiment 1

Appendix B shows the dependency diagram and code for the Ada line-oriented user interface body. The user interface operation OPEN_UI is called from the package body elaboration code to set the update interval and to display the initial elapsed and

lap time. UPDATE_TIME and UPDATE_LAP_TIME use TEXT_IO.PUT_LINE to display the current time, lap time and lap number passed in as strings from DISPLAY. Function GET_CMD uses TEXT_IO.GET to get a command from the user as a character and returns it to DISPLAY.

In this version, the elapsed time is updated when called by CHRONOGRAPH. This occurs with each start, stop, or reset command, or when the 5 second update interval expires. The lap time is updated only when lap is pressed. The line-oriented display showing this version executing is shown in Figure 4.1.

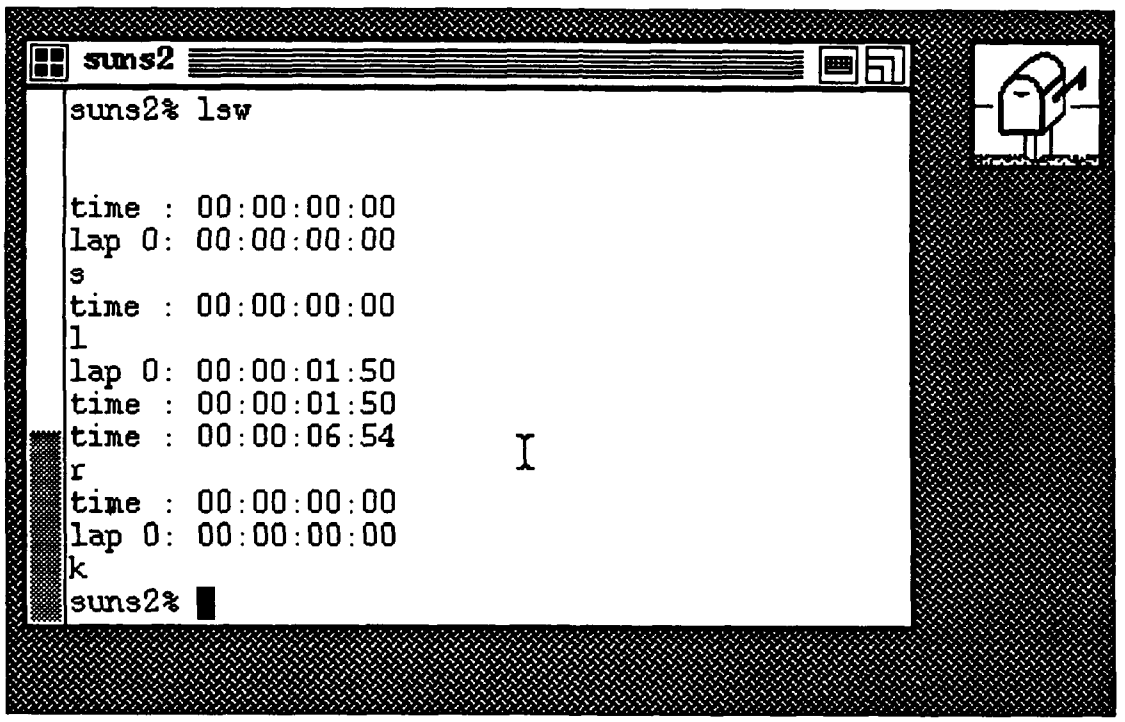

Figure 4.1 Line-oriented stop watch display in an xterm

C. CURSOR-ORIENTED USER INTERFACE

A cursor-oriented user interface was implemented using the UNIX CURSES library. Two methods were available for using CURSES: one using the VERDIX Ada development system (VADS) [Ref. 18] CURSES package, the other using the C CURSES library. The second method requires an Ada—C language interface.

Both SW versions used the following the CURSES commands[Ref. 16]:

initscr()	Initialize CURSES
noraw()	End raw input mode
echo()	Echo input mode; characters echo on screen and in window
newwin(high, width, y)	Create a new window
wrefresh(win)	Update screen to look like new window
erase()	Erase the window but don't clear screen
wgetch(win)	Get a character from a window
mvwaddstr(win, x, y, str)	Add a string to a window by calling addch()
wmove(win, x, y)	Move logical cursor to (x, y) in window
box(win, 'vert', 'hor')	Make a frame for the new window
delwin(win)	Delete the window
clear()	Reset window to blanks and clear screen if necessary

In both cases, the OPEN_UI procedure is invoked from the user interface package body to initialize the CURSES window. This requires defining the window size and location using *newwin*, moving the time and lap string into the window using *mvwaddstr*, and updating the window display using *wrefresh*. The procedure UPDATE_TIME and UPDATE_LAP_TIME put the time, lap time string, and lap number to the CURSES window and calls *wrefresh* to update the window. The procedure GET_COMMAND gets a command from the window using *wgetch*. The procedure

CLOSE_UI is provided to DISPLAY and when called terminates the window using *delwin*.

Experiment 2

Using the VADS CURSES package program, the above mentioned CURSES operations are performed directly by the user interface operations OPEN_UI, UPDATE_TIME, UPDATE_LAP_TIME, GET_COMMAND, and CLOSE_UI. Appendix C gives the package body code and Figure C.1 the dependency diagram. The SW displayed by this program is shown in Figure 4.2.

Experiment 3

Rather than use CURSES directly from Ada, this experiment shows what might be done if no Ada binding were available. In this case the user interface written in C

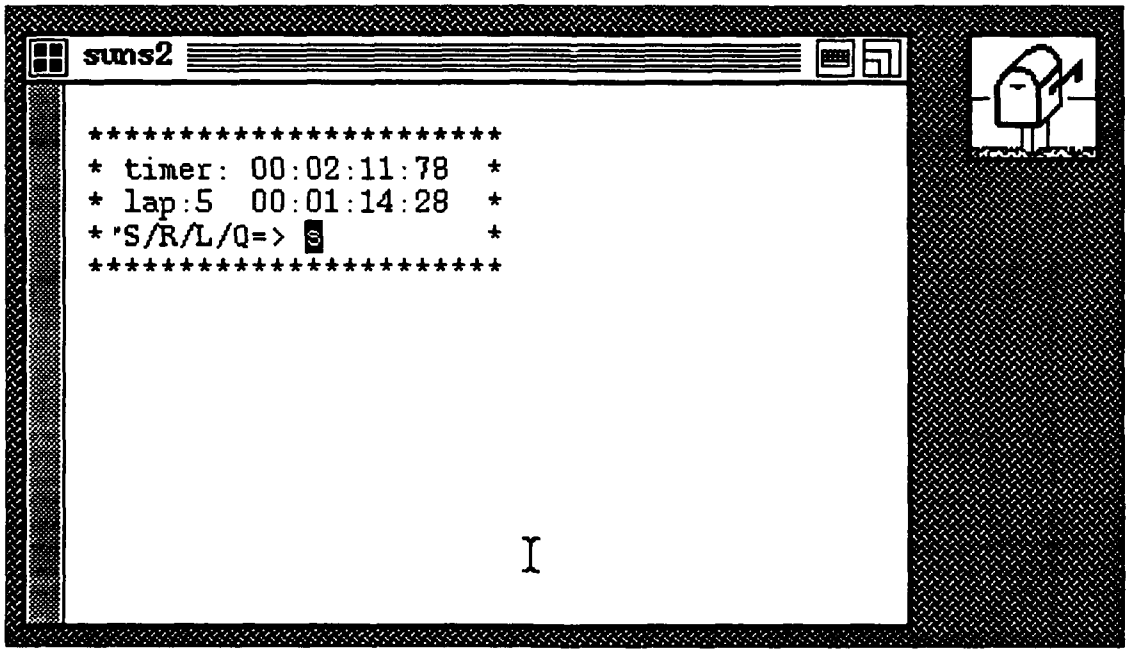

Figure 4.2 Cursor-oriented using Ada CURSES package display stop watch

provides the OPEN_UI, CLOSE_UI, GET and UPDATEs needed by display. The Ada USER INTERFACE package merely provides hooks to these C operations and declares objects for passing data between the application and the user interface. Passing parameters between Ada and C can be difficult and unreliable. Consequently, data is shared via a global data structure using pragma INTERFACE_OBJECT.

When the OPEN_UI is called, it calls the C *open* to create and initialize the timer window. GET_CMD calls the C *get_command* function which actually uses CURSES to get a character and put it in the global data structure. When control is returned to GET_CMD, that character is retrieved from the global data structure and is returned to DISPLAY. The UPDATE routines similarly put their strings in the shared data structure and call the C *display* operation which updates the window and refreshes the screen. If the command is 'k', DISPLAY calls CLOSE_UI which calls C *close* procedure to terminate the window. Appendix D shows the dependency diagram and code for the user interface package body and the C program. This program display is similar to Figure 4.2.

D. WINDOW-ORIENTED USER INTERFACE

A window-oriented user interface for SW was implemented using an InterViews C++ program. It displays the SW elapsed and lap times and provides buttons which the user selects with a mouse to execute the chronograph commands. The InterViews user interface is shown in Figure 4.3 .

The InterViews user interface is implemented as a new class *TimerWin* which is a subclass of the InterViews interactive object *Monoscene*. A *Monoscene* is a sub-

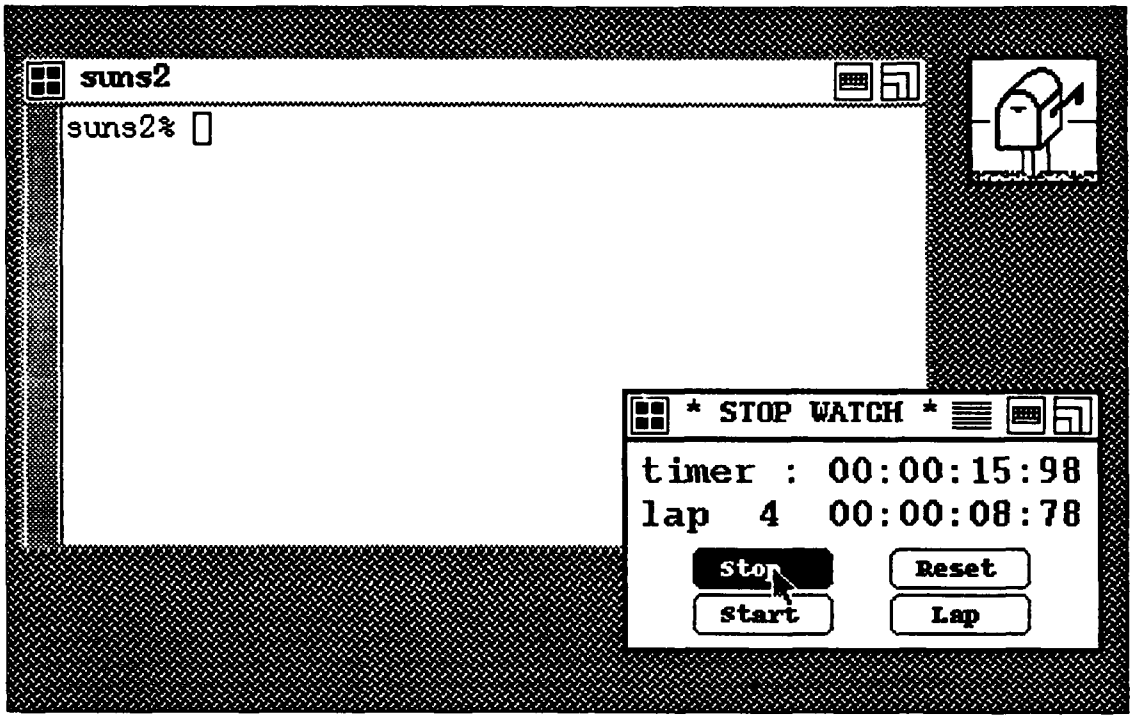

Figure 4.3 InterViews stop watch display

class for *Scene*, which is the base class for all *Interactors* that are defined in terms of one or more instances of other *Interactors*. *MonoScenes* only contain a single instance[Ref. 20]. *Scene* and *MonoScene* provide operations for handling events, displaying *Interactor* objects, and updating views.

Interactor is the base class for all interactive objects. Every *Interactor* has a shape member variable that defines the desired characteristics of screen space in terms of size, shrinkability, and stretchability. This information is used to allocate display space for the *Interactor* and it's components[Ref. 20].

Class *TimerWin* is defined to include an instance constructor operation as well as *Run*, *Update*, and *Handle*. It also defines the *ButtonState* object *press*, and *Mes-*

sage objects *time_msg*, *lap_time_msg*, and *lap_num_msg*. The *ButtonState* is a class of objects that contain a value and a list of buttons that can set its value. The *Message* is an *Interactor* that contains a line of text. The alignment of the text with respect to the *Interactor's* canvas can be specified in the constructor[Ref. 20].

TimerWin is the constructor for the class. The constructor for a class is automatically called when an instance is declared. *TimerWin* initializes the event sensors, display objects, and window structure. The *Sensor* specifies a set of input events the user interface needs to process. In this case, *key*, *button*, and *timer* events are required. Display objects include the *time_msg*, *lap_time_msg*, and *lap_num_msg Message* representing chronograph times and as well as the four *Buttons*: *start*, *stop*, *reset*, and *lap*.

The display objects are arranged in a *Box* which is a scene of *Interactors* that are tiled side-by-side in the available space. *Interactors* are placed left-to-right in a horizontal box, and top-to-bottom in a vertical *Box*. A *Box* will try to stretch or shrink the interactors inside, it to fit the available space. The *Insert*, operator is used to append an interactor to the box. Components of an *Hbox (Vbox)* will appear left-to-right (top-to-bottom) in the order in which they are inserted, Between *Boxes*, *Glue* is used to give variable-size space and allow the window to stretch or shrink[Ref. 20].

The *Run* operation is an inherited operation from *Interactor* which sets up an event loop. The event loop processes incoming events and calls event handlers for the specific interactive object to which they pertain. *Run* operation is modified for *TimerWin* so that when *UpEvents* occur for one of the four buttons, a special handle routine is called to convert the button state value to a character and put it into the shared data structure.

Update is a new operation added to change the display times in the window when they are update by the application.

The *Handle* operation is modified to process only key events. If the key pressed is 'q' (for quit—a standard InterViews termination technique) then 'k' is entered in the shared data structure command variable and the user interface is terminated.

The InterViews man program *disp_interviews* begins by creating an instance of a *World* which is the application's root scene. An instance of *TimerWin* is created and inserted to the *World*. Finally, the *TimerWin Run* event loop is called which executes the user interface until a 'q' is pressed.

The following 3 experiments compare the different control model implementation for the SW. The Ada—C++ interface is similar to the C-CURSES program (Experiment 3). A global data structure is declared to pass commands and new times between the application and the user interface.

Experiment 4

In Figure 4.4 the SW is displayed using InterViews with application-controlled control model. The user interface package body declares the needed object; operations, and links to the InterViews user interface.

In this example, the user interface operates like a dialog box. That is, when the application needs to get information, the window is displayed with the current time and the user responds by selecting a button or typing 'q' to quit. The window closes, the command is processed, and a new dialog is opened. This technique is not well suited to this example, but is valuable for pedagogical purposes. Appendix E shows

the dependency diagram for the application-controlled version and code for the Ada USER_INTERFACE package body and the C++ InterViews program.

The InterViews main program is split into two subprograms for this example. The first is called open and creates the World for the application. It is only called once and is linked to OPEN_UI in the user interface body. The second part creates a Timer-Win instance, inserts it into the World and runs it. It represents the individual dialogs and is called from GET_CMD. When it exits, GET_CMD retrieves the command from the global data structure and returns it to DISPLAY.

The UPDATE procedures merely put the text strings in the global data structure for display when the dialog is called by GET_CMD. CLOSE_UI is not required and is null.

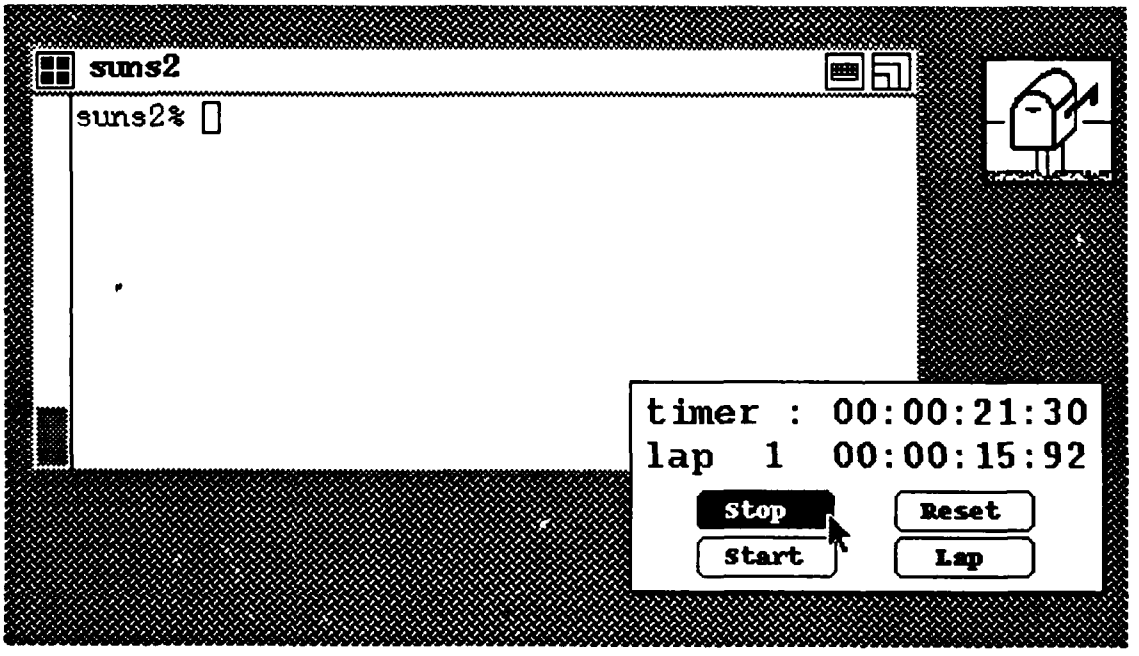

Figure 4.4 Window-oriented, application-controlled stop watch display

Experiment 5

In this example, the roles are reversed and the user interface acts as the main program. This version provides a consistent user interface but fragments the application. It calls the application operations directly to manipulate the chronograph state. This entailed a much more extensive change than previous examples. Appendix F shows the dependency diagram in Figure F.1 and the modified code. The display of this example is the same as Figure 4.3.

This model eliminates the possibility of using Ada tasks, so the TIMER task in CHRONOGRAPH is changed to five procedures: START, STOP, RESET, LAP, and UPDATE. Each procedure carries out the function of the original timer entry call and update is added to handle time event updates. Those five procedures are directly called by the user interface program using the VADS EXTERNAL_NAME pragma declared in the CHRONOGRAPH specification. The loop in the CHRONOGRAPH body used to get a command and call the TIMER entries ia not needed in this version and is deleted.

DISPLAY no longer needs a GET routine or an UPDATE_INTERVAL and they are deleted leaving only the two PUT operations. The PUTs work as before to convert the display times to strings and call USER_INTERFACE to update the global data structure.

USER_INTERFACE also delted the GET_CMD and UPDATE_INTERVAL as well as CLOSE_UI. UPDATE_TIME, and UPDATE_LAP still update the global data structure when called by the CHRONOGRAPH PUTs. In the body of USER INTER-

FACE, the TIMER_STRUCT global data structure is still defined as linking to the user interface program. OPEN_UI is linked to the user interface main program. This procedure's purpose is to start the user interface InterViews program which then acts as the main program.

The InterViews user interface program is only slightly modified to define the external names of the five Ada procedures; *start, stop, reset, lap, and update.* These are now called directly by *handle* when an UpEvent in a button is detected by *Run*, rather than have *handle* simply put a command in the shared data structure. *Update* is called only when the event is a TimerEvent.

The InterViews main program, *disp_interviews*, is called by USER_INTERFACE.OPEN_UI. It works as described at the beginning of this section to create the user interface and start its event loop.

Experiment 6

The asynchronous control model lets the application and user interface run in parallel and communicate with each other using shared memory. This experiment uses UNIX fork and shared memory system commands to set up the shared memory for asynchronous communication.

Appendix G shows the dependency diagram and software architecture for this experiment. Rather than link the Ada and C++ program together, they are linked separately with C utilities which handle the interprocess communication(IPC). This separates the system specific IPC protocol from application and user interface functionality.

The USFR INTERFACE package UPDATEs and GET operations are implemented the same as with application-controlled experiment. The OPEN_UI operation is changed to call the C programs *open_shared_memory* and *spawn_UI*. The *open_shared_memory* operation gets and attaches the shared memory area using UNIX system commands *shmget* and *shmat* as discussed in Chapter III, section C.3.c. *Spawn_UI* uses the system *fork* and *execlp* commands to create and start the InterViews main program as a separate process. CLOSE_UI calls *close_shared_memory* using the system command *shmctl* to release the shared memory resource.

The application C utility linked with the Ada program defines the C data structure and contains the three functions *open_shared_memory*, *spawn_UI*, and *close_shared_memory*. The user interface C utility links the user interface to the shared memory and is linked with the InterViews program. The utility *main* procedure attaches to same shared memory area as the application program and calls the InterViews main program *disp_interviews*. The application and user interface now may pass information through the shared memory global data structure as in example 4. The user interface display is same as Figure 4.3.

In the SW asynchronous version, the busy/wait concept is used to avoid a race condition and window flicker. The race condition may occur when the user interface accepts commands faster than the application handles them. This is prevented by using a ' ' (space) command value. To the application, a ' ' command tells it to wait for a command to be given. When finished processing a command, the USER INTERFACE package body GET_CMD routine resets the command to ' '. To the InterViews program, just the opposite is true. When the command is non-blank, the user interface

waits until the application is finished with the current command (command reset to ' ') before putting the new command in the global data structure. This busy/wait scheme emulates the behavior of the Ada rendezvous.

Display flicker is caused by the user interface updating the display more frequently than the data to be displayed is updated. This is eliminated by associating a boolean flag with the elapsed and lap time strings. This flag is set when the application updates the global data area and reset when the display is redrawn. In this case, data can be lost when the application gets a head of the user interface. This illustrates the real-time situation when only current information is of value.

E. EXECUTION OF APPLICATIONS IN X

Additional considerations when building X user interfaces include how the application program is executed and where the user interface is visually located on the screen. Programs which require a user interface may be executed via the window manager, command line, or by selecting an icon. Experiments 7-9, illustrate these techniques.

Experiment 7

Window managers like twm [Ref. 21] provide the capability to define menus which can be used to invoke programs. Menus can also refer to other menus providing layered access to applications, tools, and utility functions. Menus are defined in the window manager's initialization file (see window displayed in Figure 4.5) by giving a menu name, a list of menu entries, and a corresponding list of commands. Menu names are tied to a keystroke/mouse button/window context for activation.

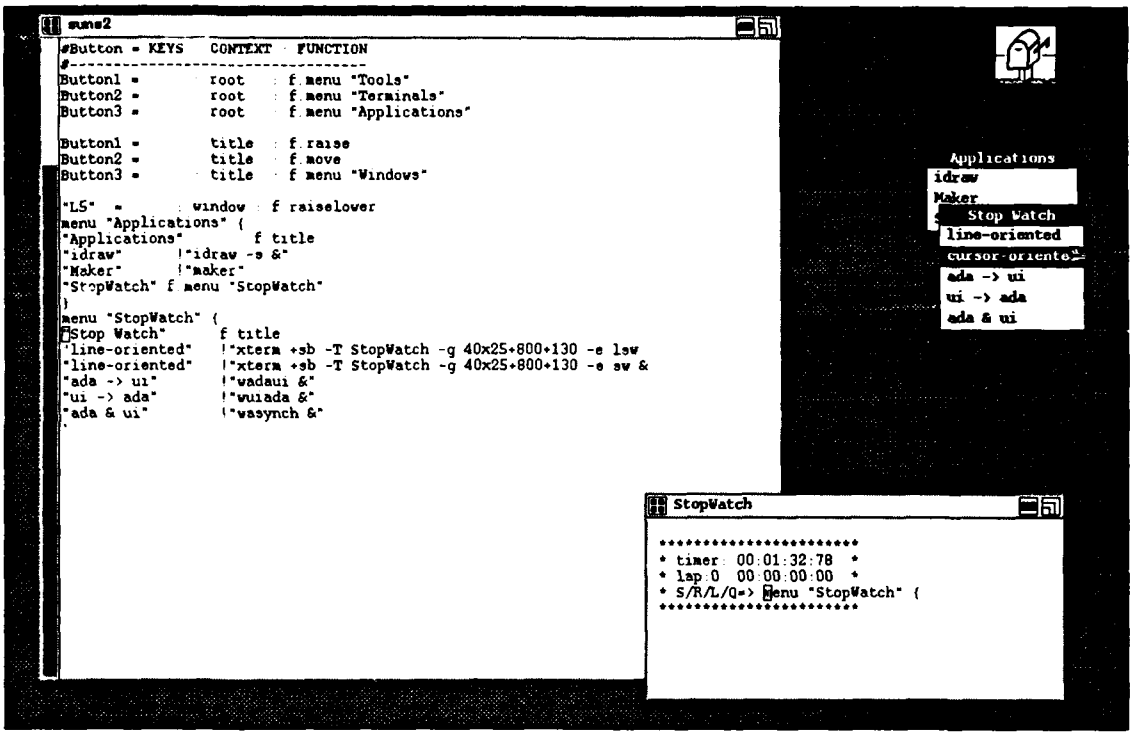

Figure 4.5 Executing applications using menus

Whenever the right combination of keys and mouse buttons is pressed and the mouse pointer is in the right context (root, title, window, icon, or frame) the menu is activated. In Figure 4.5, the mouse pointer is in the root and the third mouse button is pressed causing the menu with a list of application names to appear. The StopWatch was selected by moving the cursor to that entry arrow causing a new window to appear listing examples. The selected application is the one under the mouse pointer and is shown in reverse video. When the mouse button is released, the corresponding command is executed to start the application. Note that when a program is executed via the window manager, the I/O is through the same window which executed the

window manager. Consequently, application are normally executed their own xterm (note the line-oriented example command code shown in the window in Figure 4.5).

Experiment 8

One problem with using the window manager menu is that the command line is fixed in the window manager initialization file (see Figure 4.5). To give the user more control over application options, the application may be executed directly via command.

In an xterm, the user can type the command line to execute the application giving a variety of X windows options such as location, window size, font etc. A command line example is shown in Figure 4.6.

In this example, an xterm is executed with a title (-T), size and position (-g), no

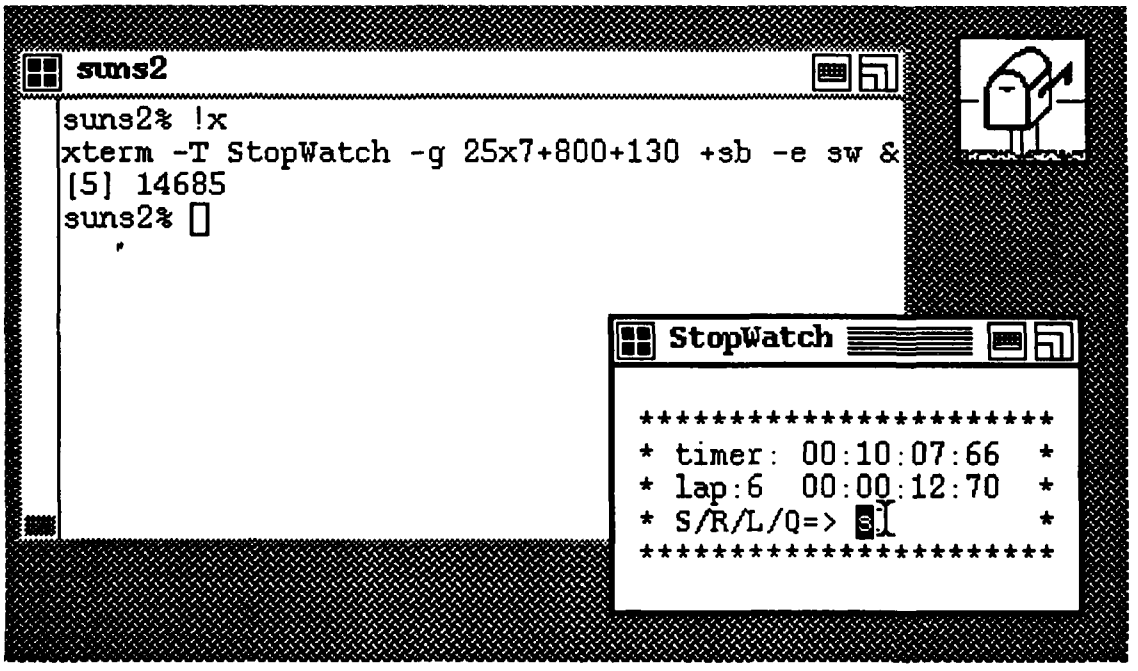

Figure 4.6 Command line to execute applications or icon

scroll bar (+sb) and will run the SW program in the new xterm window (-e). The '&' tells the xterm to run as a background task. For more detail see[Ref. 21].

Experiment 9

To provide a super simple means of executing an application by only selecting an icon, a variation of the InterViews *Logo* program was developed. This program accepts as options the name of an application and a bitmap to display. Running *bmx* (bitmap executor) the program causes the named bitmap to be displayed. When the mouse button is clicked in the icon, an instance of the named application is executed. As a default, the xterm application is executed. See Figure 4.7. Appendix H lists the *Logo* modules which changed for *Bmx*. These methods that changed only by changing the class name from the Logo to Bmx are not shown.

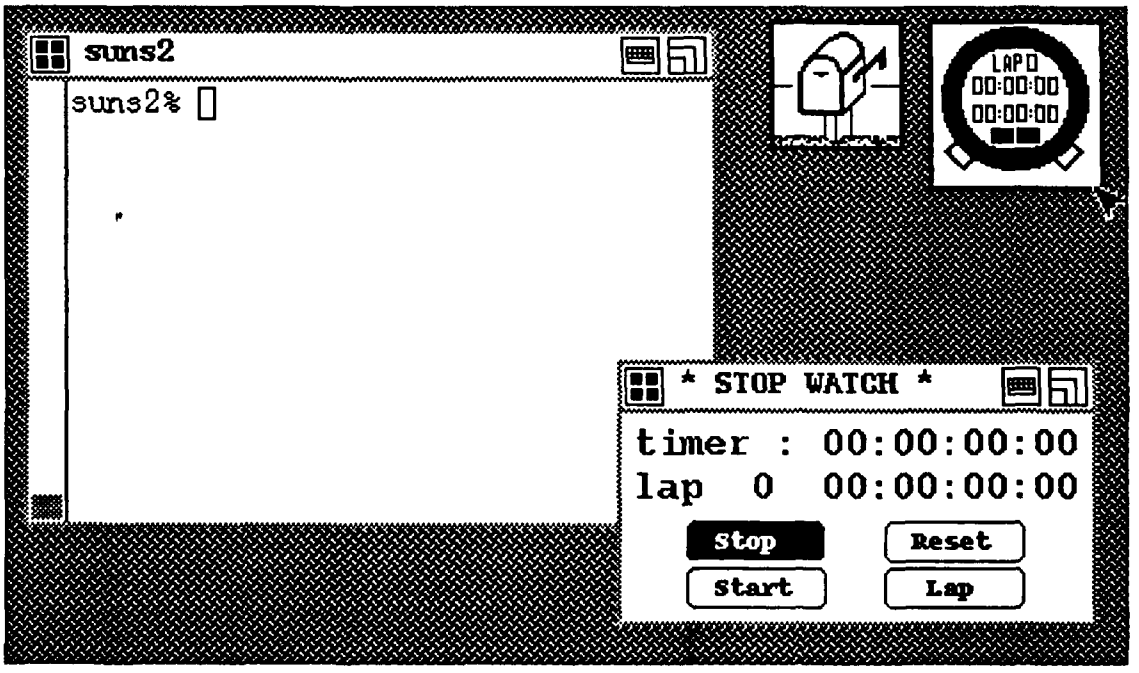

Figure 4.7 Application executed by an icon

V. CONCLUSION

A. SUMMARY

This thesis presented the user interface problem for Ada real-time programs and discussed several solutions given current technology. The following step-by-step approach is provided for using these results:

1. Determine the I/O Model

First, list tasks to be performed by the user. Analyze tasks and decide which user interface I/O model is appropriate. Skip to step 2, 3 or 4 depending on I/O model.

2. Define Line-Oriented I/O

Storyboard system prompts and user responses. Implement using TEXT_IO. Proceed to step 5.

3. Define Cursor-Oriented I/O

Story board screen displays and implement in curses or some other cursor-oriented Ada package. If none is available, write user interface in C and define the set of objects, type, and subprograms for the Ada interface. Skip to step 5.

4. Define Window-Oriented I/O

Story board dialog, graphics, and other interactive windows. Implement user interface displays using X toolkit widgets.

5. Determine the Control Model

Analyze user task/application interaction and identify which control model to use. Skip to step 6, 7, or 8 depending on which control model is appropriate.

6. Describe Application Control

Map application user/user interface requirements to user interface prompts, screen displays, or dialogs. Determine linker names for user interface subprograms and objects and design necessary parallel data types. Proceed to step 9.

7. Describe User Interface Control

Map user interface, control switches, buttons, and menu selection to application routines. Determine Ada subprogram names, data types objects for access by the user interface. Proceed to step 9.

8. Describe Asynchronous Control

Determine required shared memory data structure and model in both Ada and C. Tailor C utilities to these data structures to initialize and close shared memory segment and fork user interface. Analyze deadlock, data integrity possibilities and implement semaphore control it if necessary.

9. Define the Invocation Methods of the Program

Finally, analyze frequency of use and user experience or background to determining the best application invocation method. If command line: document user options and command line syntax. If window manager menu, then write menu code for in-

clusion in the window manager start up file (e.g. .twmrc). If it is to be started by icon, design an appropriate icon and determine how the icon generator program is to be executed—(e.g. from .xinitrc, .login, menu or command line).

B. RECOMMENDATION

The description of Ada user interface development for X workstations provides a starting point for future work developing the user interface for LCCDS. Tools are available now, techniques can be readily understood and applied. However, difficulty in learning and integrating Ada, C, C++, InterViews, X concepts suggest a simpler approach is needed for better maintainability of future CDS user interface. Ada binding to X and X toolkits are needed. A public domain Ada binding to Xlib is currently available but dubious reliability. Work is being done to bind Ada to the Xt Intrinsic(basic toolkits functions) as well as several widget sets including HP and Motif(is a trademaker of the Open Software Foundation).

Tools, such as the TAE+ Interface Workbench provide a higher level abstraction for development of user interfaces. However, they employ the same mutil-language complexities found in InterViews. In general user interface development tools for Ada are either unavailable or not yet mature enough to build a reliable user interface for CDS. A major goal for the next generation of Ada software development should include:

- binding of Ada to existing toolkits,
- develop toolkits directly in Ada,
- develop high level interactive tools for automatic user interface generation for Ada programs.

APPENDIX A

STOP WATCH PROGRAM INTRODUCTION

This Appendix contains the dependency diagram and code for the basic Stop Watch program. In addition, Figure A.1 shows the dependency diagram constructs used throughout the appendices.

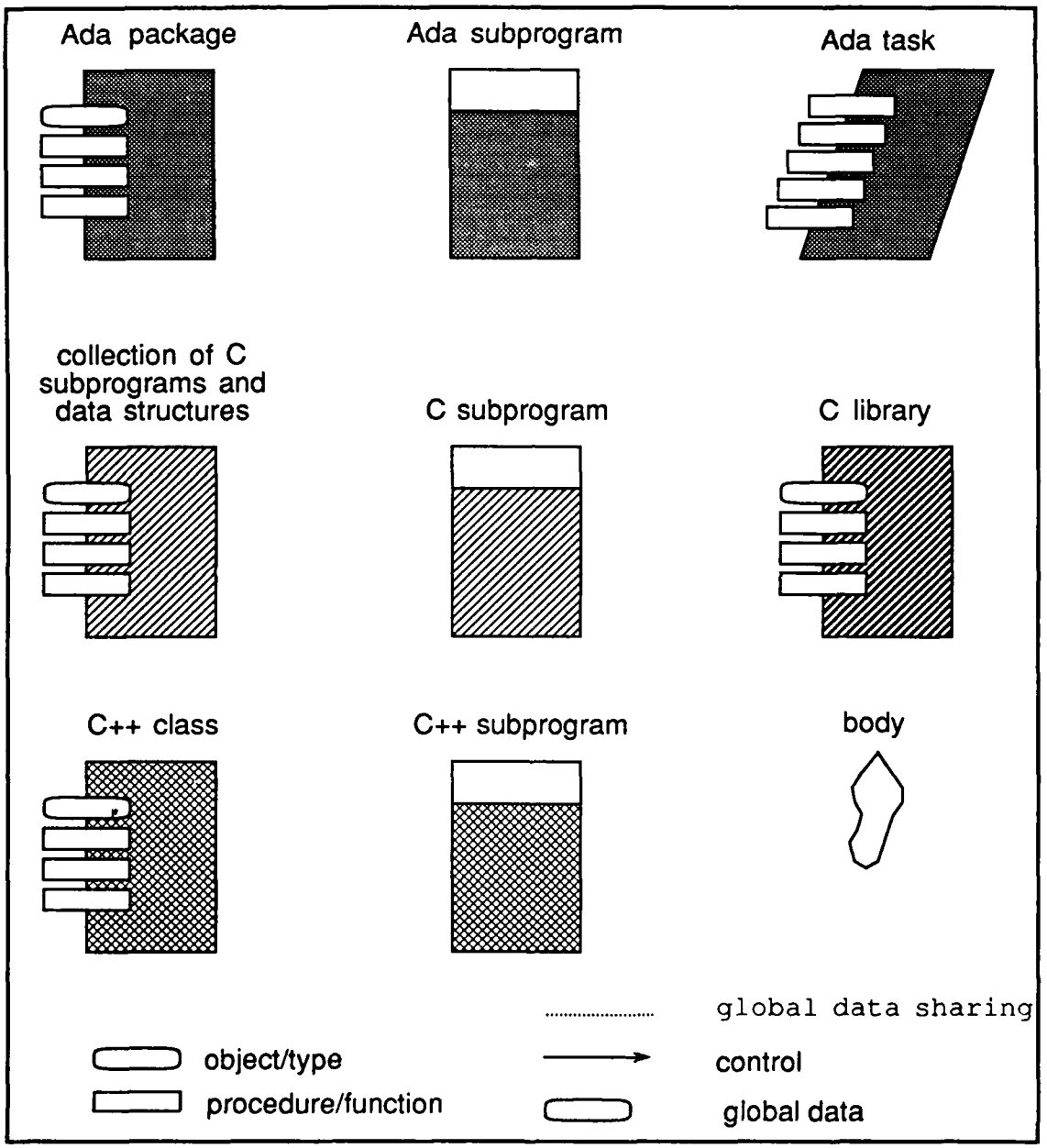

Figure A.1. Key to Ada Dependency Diagrams

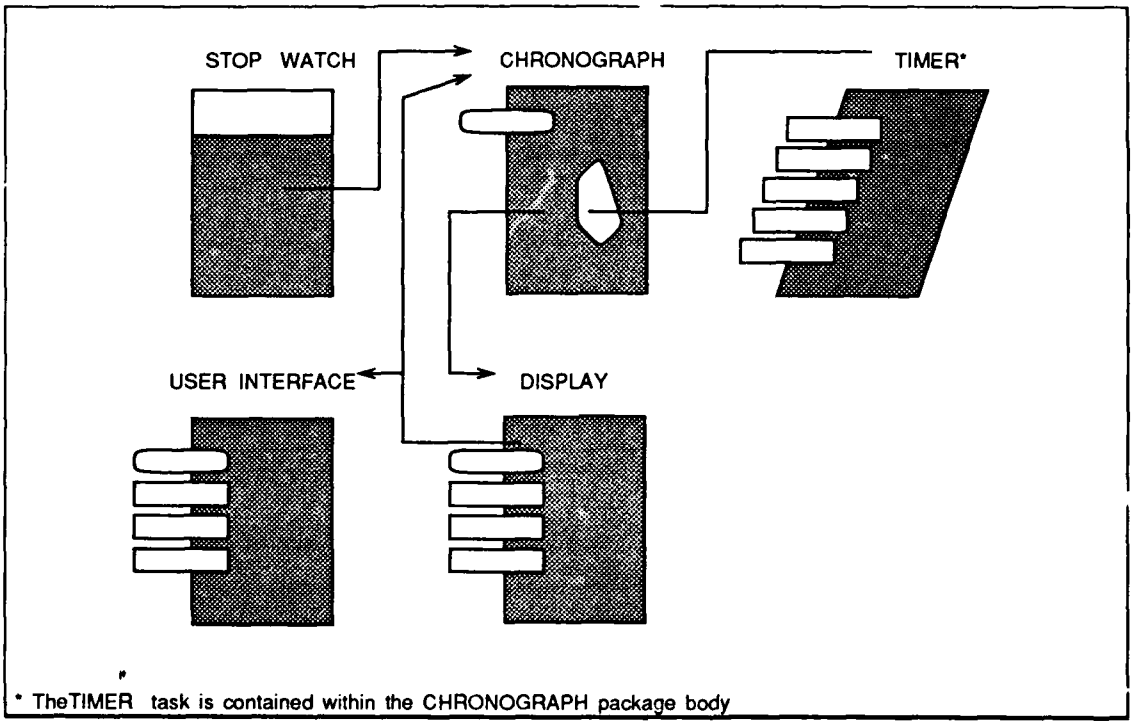

Figure A.2 Stop Watch Dependency Diagram

```
-------------------------------------------------------------------------
-- file        : sw.a
-- author      : sun chien-hsiung
-- subject     : Stop Watch main program.
-- description: This main program is only to start running the program.
-------------------------------------------------------------------------
with CHRONOGRAPH;
use  CHRONOGRAPH;
procedure STOP_WATCH is
begin
  null;
end STOP_WATCH;
```

```
-------------------------------------------------------------------------
-- file        : chron_s.a
-- author      : sun chien-hsiung
-- subject     : package specification for stop watch
-- description: The chronograph package defines the abstract state ma-
--     chine used in the stop watch program.  The data structure types
--     are defined for use by users of the package.
-------------------------------------------------------------------------

package CHRONOGRAPH is

  subtype NUM_LAPS   is INTEGER range 0 .. 9;
  subtype TWO_DIGITS is INTEGER range 0 .. 99;

  -- Timer_Record represents the time values to be exported to using
  -- programs
  type TIMER_RECORD is record
    HOURS       : TWO_DIGITS := 0;
    MINUTES     : TWO_DIGITS := 0;
    SECONDS     : TWO_DIGITS := 0;
    HUNDREDTHS  : TWO_DIGITS := 0;
  end record;

  type COMMAND is (START, STOP, LAP, RESET, KILL);

end CHRONOGRAPH;
```

```
-------------------------------------------------------------------------
-- file         : chron_b.a
-- author       : sun chien-hsiung
-- subject      : An abstract state machine to perform stop watch
--                functions.
-- Description : The chronograph contains the timer task, internal
--   elapsed time, lap times, and the timer state.  It handles user
--   requests to start, stop, reset, and record/review laps.  A kill
--   function is provided to terminate the state machine.  A DISPLAY
--   package is needed to provide get and put functions and an update
--   interval.
-------------------------------------------------------------------------
with CALENDAR, DISPLAY;
use  CALENDAR;
pragma ELABORATE(DISPLAY);
package body CHRONOGRAPH is

-- convert system time to two digit numbers
  function CONVERT_TIME(TIME : in DURATION) return TIMER_RECORD is
    INT_TIME   : INTEGER;
    TIMER_TIME : TIMER_RECORD;
  begin
    INT_TIME := INTEGER(TIME);
    TIMER_TIME.HOURS := INT_TIME / 3600;
    TIMER_TIME.MINUTES := INT_TIME mod 3600 / 60;
    TIMER_TIME.SECONDS := INT_TIME mod 60;
    TIMER_TIME.HUNDREDTHS := INTEGER(FLOAT(TIME) * 100.0) mod 100;
    return TIMER_TIME;
  end CONVERT_TIME;

  -- Task entry calls for stop watch functions
  task TIMER is
    entry START;
    entry STOP;
    entry LAP;
    entry RESET;
    entry KILL;
  end TIMER;

  -- The task represents an abstract state machine, updating the
  -- clock value in real-time and processing user commands.
  task body TIMER is
    type LAP_ARRAY is array(NUM_LAPS) of DURATION;
    type STATES    is (INITL, RUNNING, STOPPED);
    ELTIME            : DURATION     := 0.0;
    DISP_LAP          : NUM_LAPS     := NUM_LAPS'FIRST;
    LAPS              : LAP_ARRAY    := (others => 0.0);
    NEXT_LAP          : NUM_LAPS     := NUM_LAPS'FIRST;
    STATE             : STATES       := INITL;
    START_TIME        : TIME         := CLOCK;
    EL_TIME, LAP_TIME : TIMER_RECORD := (others => 0);
```

```
begin
-- Begin timer entry call.
-- If no rendezvous is ready within system tick
-- seconds, rendezvous is canceled.
  loop
    select
      accept START do     -- Start change state to running
        case STATE is
          when INITL =>
            START_TIME := CLOCK;
          when STOPPED =>
            START_TIME := CLOCK - ELTIME;
          when others =>
            null;
        end case;
        STATE := RUNNING;
        EL_TIME := CONVERT_TIME(ELTIME);
        DISPLAY.PUT(EL_TIME);
      end START;
    or
      accept STOP do
        case STATE is
          when RUNNING =>
            ELTIME := CLOCK - START_TIME;
            STATE := STOPPED;
          when others =>
            null;
        end case;
        EL_TIME := CONVERT_TIME(ELTIME);
        DISPLAY.PUT(EL_TIME);
      end STOP;
    or
      accept LAP do
        case STATE is
          when RUNNING =>
            if NEXT_LAP <= NUM_LAPS'LAST then
              DISP_LAP := NEXT_LAP;
              LAPS(DISP_LAP) := CLOCK - START_TIME;
              if NEXT_LAP < NUM_LAPS'LAST then
                NEXT_LAP := NUM_LAPS'SUCC(DISP_LAP);
              end if;
            end if;
          when STOPPED =>
            if NEXT_LAP > NUM_LAPS'FIRST and then DISP_LAP <
             NUM_LAPS'PRED(NEXT_LAP) then
              DISP_LAP := NUM_LAPS'SUCC(DISP_LAP);
            else
              DISP_LAP := NUM_LAPS'FIRST;
            end if;
```

```
            when others =>
              null;
          end case;
          LAP_TIME := CONVERT_TIME(LAPS(DISP_LAP));
          DISPLAY.PUT(LAP_TIME, DISP_LAP);
        end LAP;
      or
        accept RESET do
          case STATE is
            when STOPPED | RUNNING =>
              NEXT_LAP := LAP_ARRAY'FIRST;
              DISP_LAP := LAP_ARRAY'FIRST;
              LAPS := (others => 0.0);
              STATE := INITL;
              ELTIME := 0.0;
            when others =>
              null;
          end case;
          EL_TIME := CONVERT_TIME(ELTIME);
          LAP_TIME := CONVERT_TIME(LAPS(DISP_LAP));
          DISPLAY.PUT(EL_TIME);
          DISPLAY.PUT(LAP_TIME, DISP_LAP);
        end RESET;
      or
        accept KILL;
        exit;
      or
        delay DISPLAY.UPDATE_INTERVAL;
      end select;
      if STATE = RUNNING then
        ELTIME := CLOCK - START_TIME;
        EL_TIME := CONVERT_TIME(ELTIME);
        DISPLAY.PUT(EL_TIME);
      end if;
    end loop;
  end TIMER;
```

```
begin     -- PACKAGE CHRONOGRAPH
          -- This package body statement section represents an Ada pro-
          -- cess which handles user input asynchronously with the ASM
          -- Timer.
  loop
    case DISPLAY.GET is
      when START =>
        TIMER.START;
      when STOP =>
        TIMER.STOP;
      when LAP =>
        TIMER.LAP;
      when RESET =>
        TIMER.RESET;
      when KILL =>
        TIMER.KILL;
        exit;
      when others =>
        null;
    end case;
  end loop;
end CHRONOGRAPH;
```

```
-------------------------------------------------------------------------
--- file        : disp_s.a
-- author      : sun chien-hsiung
-- subject     : This package represents the system display device.
-- description: Get and put operations are provided for updating time
--     and lap time/number and for retrieving user input commands. A
--     delay time is provided for suggesting a minimum delay between
--     updates depending on user-interface type.
-------------------------------------------------------------------------
-with CHRONOGRAPH;
use  CHRONOGRAPH;

package DISPLAY is

  UPDATE_INTERVAL : DURATION;
  procedure PUT(EL_TIME : in TIMER_RECORD);
  procedure PUT(LAP_TIME : in TIMER_RECORD; LAP : in NUM_LAPS);
  function GET return COMMAND;

end DISPLAY;
```

```
------------------------------------------------------------------------
--- file        : disp_b.a
-- author      : sun chien-hsiung
-- subject     : implements I/O for the Stop Watch program.
-- description: I/O is limited to implementation hidden in user inter-
--     face package.  On output, values received from the Timer ASM are
--     converted to strings and the UI package routines are called. Like
--     wise, commands are retrieved in character form and converted to
--     the enumerated type specified by the CHRONOGRAPH package. The
--     Update interval provided in the specification is initialized
from
--     the user interface since it is dependent on user interface type.
------------------------------------------------------------------------
-with USER_INTERFACE;
pragma ELABORATE(USER_INTERFACE);
package body DISPLAY is

-- convert the integer time to string

  function INT_TO_STRING(NUMBER, WIDTH : in INTEGER) return STRING is
    NEW_STRING : STRING(1 .. WIDTH) := (others => '0');
    DIGIT      : INTEGER;
    NEW_NUMBER : INTEGER             := NUMBER;
  begin
    for I in reverse 1 .. WIDTH loop
      DIGIT := NEW_NUMBER mod 10;
      NEW_STRING(I) := CHARACTER'VAL(DIGIT + 48);
      NEW_NUMBER := NEW_NUMBER / 10;
    end loop;
    return NEW_STRING;
  end INT_TO_STRING;

  -- put times(hh,mm,ss,ms) to as a display string

  function TIME_TO_STRING(TIME : in TIMER_RECORD) return STRING is
    NEW_STRING : STRING(1 .. 11) := "00:00:00:00";
  begin
    NEW_STRING(1 .. 2)  := INT_TO_STRING(TIME.HOURS, 2);
    NEW_STRING(4 .. 5)  := INT_TO_STRING(TIME.MINUTES, 2);
    NEW_STRING(7 .. 8)  := INT_TO_STRING(TIME.SECONDS, 2);
    NEW_STRING(10 .. 11) := INT_TO_STRING(TIME.HUNDREDTHS, 2);
    return NEW_STRING;
  end TIME_TO_STRING;

  -- put display time string to User_Interface.

  procedure PUT(EL_TIME : in TIMER_RECORD) is
  begin
    USER_INTERFACE.UPDATE_TIME(TIME_TO_STRING(EL_TIME));
  end PUT;
```

```
   -- put lap time string

  procedure PUT(LAP_TIME : in TIMER_RECORD; LAP : in NUM_LAPS) is
  begin
    USER_INTERFACE.UPDATE_LAPS(TIME_TO_STRING(LAP_TIME),
                                          INT_TO_STRING(LAP,1));
  end PUT;

  -- get a command from the user
  function GET return COMMAND is
  begin
        begin
      loop
        case USER_INTERFACE.GET_CMD is
          when 's' =>
            return START;
          when 'q' =>
            return STOP;
          when 'l' =>
            return LAP;
          when 'r' =>
            return RESET;
          when 'k' =>
            USER_INTERFACE.CLOSE_UI;
            return KILL;
          when others =>
            null;
        end case;
      end loop;
    end;
  end GET;
begin
  DISPLAY.UPDATE_INTERVAL := USER_INTERFACE.UPDATE_INTERVAL;
end DISPLAY;
```

```
-------------------------------------------------------------------------
-- file        : ui_s.a
-- author      : sun chien-hsiung
-- subject     : User interface implementation for Stop Watch program.
-- description: The user interface provides implementation specific
--    details depending on style of user interface.  The specification,
--    however, does not change, only the body.  Various types of user
--    interfaces include:
--    line-oriented
--    cursor-oriented
--    window-oriented
-------------------------------------------------------------------------

with SYSTEM;

package USER_INTERFACE is
  UPDATE_INTERVAL : DURATION := SYSTEM.TICK;
  procedure CLOSE_UI;
  procedure UPDATE_TIME(TIME : in STRING);
  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING);
  function GET_CMD return CHARACTER;
end USER_INTERFACE;
```

```
-------------------------------------------------------------------------
-- file        : ui_b.a
-- author      : sun chien-hsiung
-- subject     : xxxx-orinted user interface implementation for Stop
--               Watch.
-- description: This is the template for user interface bodies.It
--    will be different for each style/type of user interface.
-------------------------------------------------------------------------

package body USER_INTERFACE is

  procedure OPEN_UI is
  begin
    null;
  end OPEN_UI;

  procedure CLOSE_UI is
  begin
    null;
  end CLOSE_UI;

  procedure UPDATE_TIME(TIME : in STRING) is
  begin
    null;
  end UPDATE_TIME;

  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING) is
  begin
    null;
  end UPDATE_LAPS;

  function GET_CMD return CHARACTER is
    CH : CHARACTER := 'q';
  begin
    return CH;
  end GET_CMD;
begin
  OPEN_UI;
  UPDATE_INTERVAL := 1.0;
end USER_INTERFACE;
```

APPENDIX B

LINE-ORIENTED USER INTERFACE USING TEXT.IO

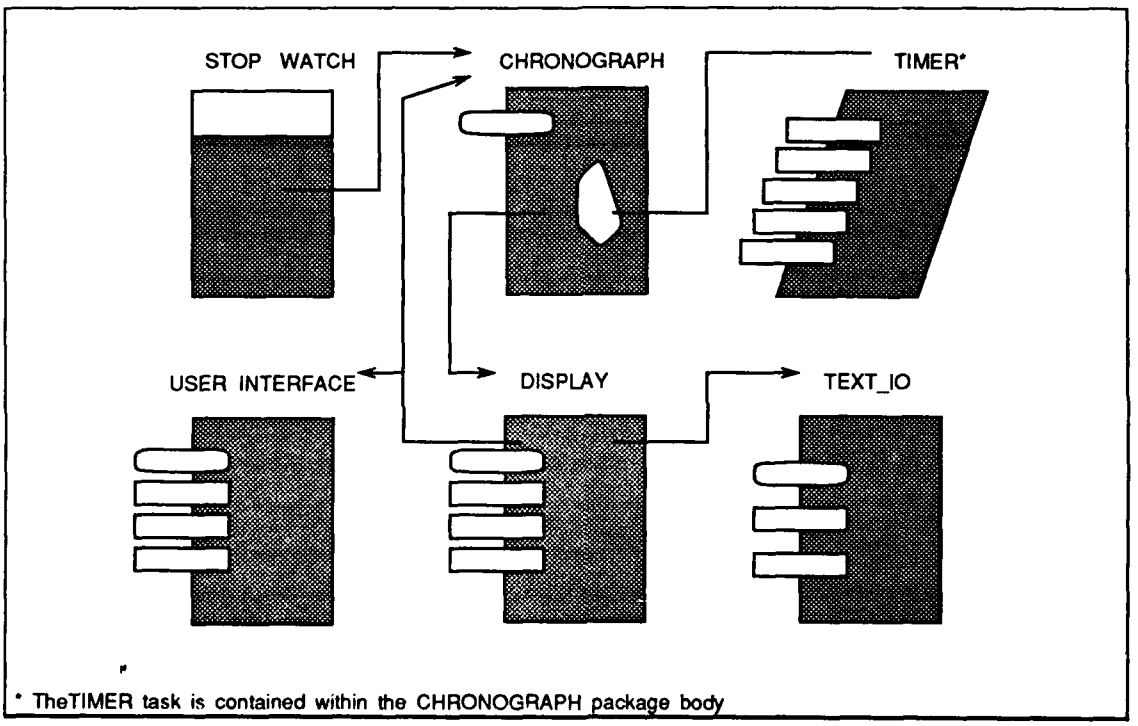

Figure B.1 Line-oriented User Interface Using TEXT.IO

```
-------------------------------------------------------------------------
--- file        : ui_b.a
-- author      : sun chien-hsiung
-- subject     : Line-orinted user interface implementation for Stop
--               Watch.
-- description: The line-oriented user interface uses the Ada TEST_IO
--    package to provide puts and gets of text strings from the termi-
--    nal. Each output line includes the appropriate constant "Time:"
--    or "Lap:" and input is in the character format DISPLAY re-
quires.
--    The UPDATE_INTERVAL is initialized to 5 sec so the output is
--    manageable.
-------------------------------------------------------------------------
-with TEXT_IO;
package body USER_INTERFACE is
  -- to initialize the timer display
  procedure OPEN_UI is
  begin
    UPDATE_INTERVAL := 5.0;
    TEXT_IO.NEW_PAGE;
    UPDATE_TIME("00:00:00:00");
    UPDATE_LAPS("00:00:00:00", "0");
  end OPEN_UI;
  procedure CLOSE_UI is
  begin
    null;
  end CLOSE_UI;
  -- display the elapsed time
  procedure UPDATE_TIME(TIME : in STRING) is
  begin
    TEXT_IO.PUT_LINE("time : " & TIME);
  end UPDATE_TIME;

  -- display the lap number and lap time
  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING) is
  begin
    TEXT_IO.PUT_LINE("lap " & LAP_NUM & ": " & LAP_TIME);
  end UPDATE_LAPS;

  -- to get command from the terminal
  function GET_CMD return CHARACTER is
    CH : CHARACTER;
  begin
    TEXT_IO.GET(CH);
    TEXT_IO.SKIP_LINE;
    return CH;
  end GET_CMD;
-- begin the user interface body and start the timer display
begin
  OPEN_UI;
end USER_INTERFACE;
```

APPENDIX C

CURSOR-ORIENTED USER INTERFACE USING ADA CURSES PACKAGE

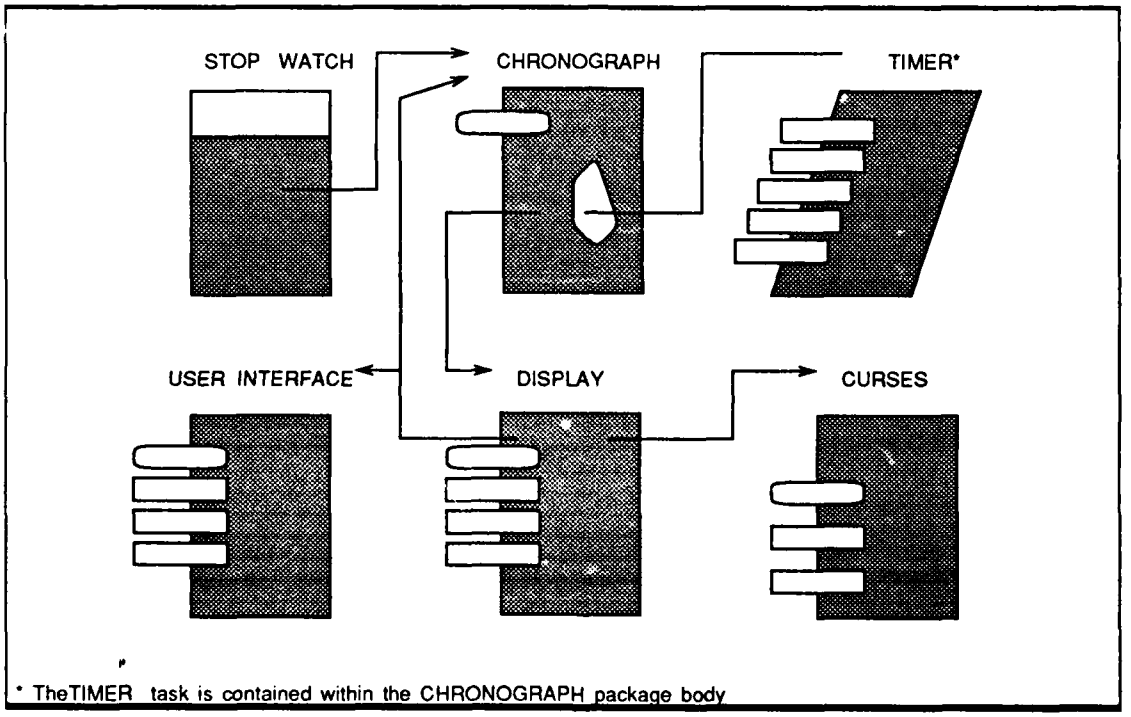

Figure C.1 Cursor-oriented User Interface Using Ada CURSES Package

```
------------------------------------------------------------------------
---- file        : ui_b.a
-- author      : sun chien-hsiung
-- subject     : cursor-orinted user interface implementation for Stop
--               Watch.
-- description: The cursor-oriented user interface uses the Ada curs-
es
--    package to display the timer. Basic functions found in curses
--    are:
-- move the cursor to any point on the screen.
-- insert text anywhere on the screen, doing it even in highlight
mode.
-- divide the screen into rectangular areas called windows.
-- manage each window independently, so we can be scrolled while an-
other
-- is being edited.
-- draw a box around a window using any character.
-- Commands that using in this program are:
--
-- newwin(high, wide, x, y) create a new window.
-- wrefresh(win) update screen to look like new window.
-- erase() erase the window but don't clear screen.
-- wgetch(win) get a character from a window.
-- mvwaddstr(win, x, y, str) add a string to a window by calling ad-
dch()
-- wmove(win, x, y) move logical cursor to (x,y) in window.
-- box(win, 'key', 'key') make a frame for the new window.
-- delwin(win) delete the window.
-- clear() reset window to blanks and clear screen if necessary.
--
-- This program open is to define the new window and display and win-
dow
-- position. Updates are put the current time , lap  time, and lap
num-
-- ber to the window and the get is to get command form the window
and
-- return to the display package. The last close is to  terminate the
-- window.
------------------------------------------------------------------------
--with CURSES;
use  CURSES;
package body USER_INTERFACE is

 -- creating the window and initializing it and refresh to open the
 -- window display the initialized timer.
  UI_WINDOW : WINDOW;
  procedure OPEN_UI is
  begin
    INITSCR;
    CLEAR;
    UI_WINDOW := NEWWIN(5, 23, 1, 1);
```

```
    BOX(UI_WINDOW, '*', '*');
    MVWADDSTR(UI_WINDOW, 1, 2, "timer: 00:00:00:00");
    MVWADDSTR(UI_WINDOW, 2, 2, "lap:   00:00:00:00");
    MVWADDSTR(UI_WINDOW, 3, 2, "S/R/L/Q=>");
    WREFRESH(UI_WINDOW);
  end OPEN_UI;
  --Close the user interface by clearing the window contents and
  --removing it from the system.

  procedure CLOSE_UI is
  begin
    CLEAR;
    REFRESH;
    NORAW;
    ECHO;
    DELWIN(UI_WINDOW);
  end CLOSE_UI;

  -- update the display time

  procedure UPDATE_TIME(TIME : in STRING) is
  begin
    MVWADDSTR(UI_WINDOW, 1, 9, TIME);
    WMOVE(UI_WINDOW, 3, 12);
    WREFRESH(UI_WINDOW);
  end UPDATE_TIME;

  -- update display lap time

  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING) is
  begin
    MVWADDSTR(UI_WINDOW, 2, 9, LAP_TIME);
    MVWADDSTR(UI_WINDOW, 2, 7, LAP_NUM);
    WMOVE(UI_WINDOW, 3, 12);
    WREFRESH(UI_WINDOW);
  end UPDATE_LAPS;

  -- get command from the window
  function GET_CMD return CHARACTER is
    CH : CHARACTER := 'q';
  begin
    CH := WGETCH(UI_WINDOW);
    return CH;
  end GET_CMD;

-- set the interval time and call to create the initial window
begin
  UPDATE_INTERVAL := 1.0;
  OPEN_UI;
end USER_INTERFACE;
```

APPENDIX D

CURSOR-ORIENTED USER INTERFACE USING C CURSES LIBRARY

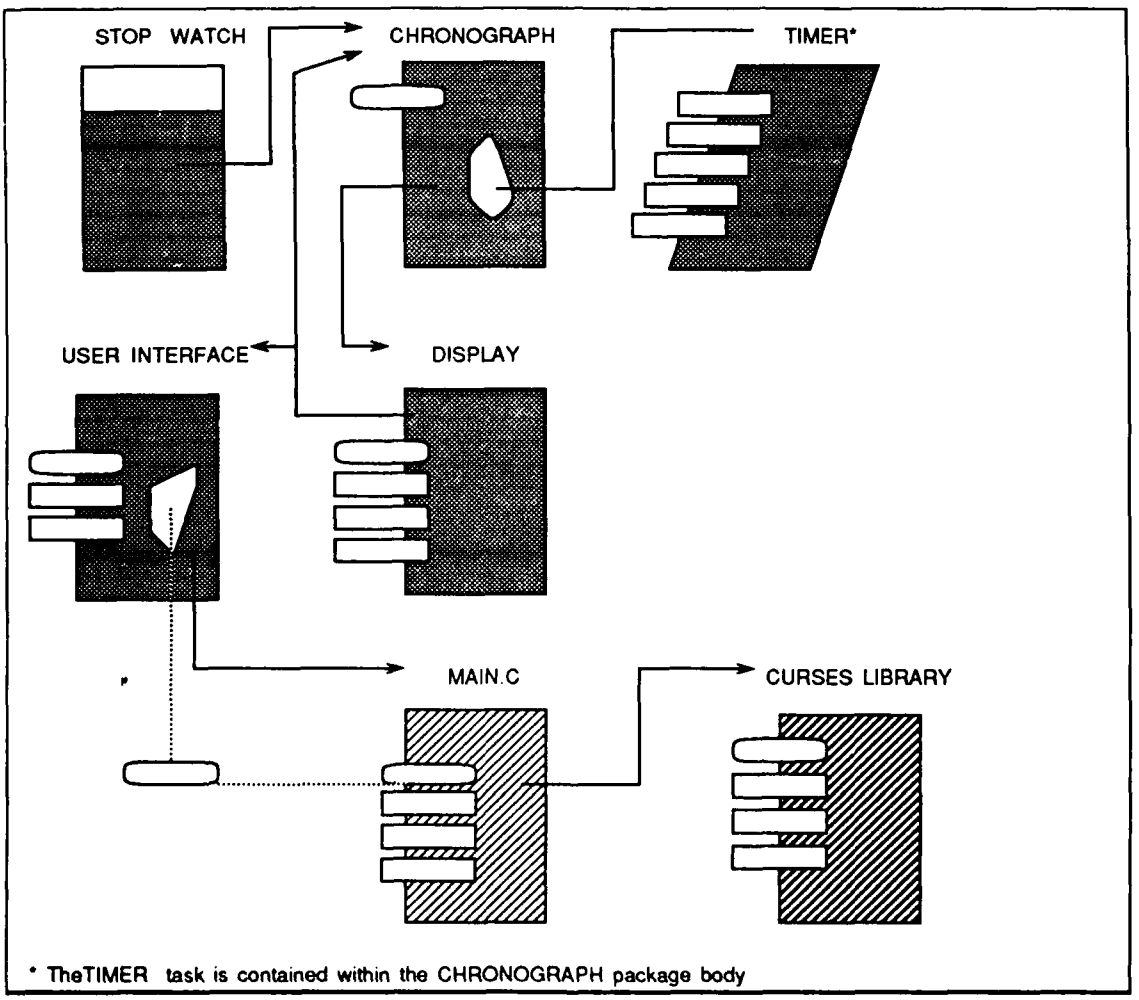

Figure D.1 Cursor-oriented User Interface Using C CURSES Library

```
-------------------------------------------------------------------------
-- file        : ui_b.a
-- author      : sun chien-hsiung
-- subject     : c-cursor-orinted user interface implementation for Stop
--               Watch
-- description: The c-cursor-oriented user interface uses a separate c
--    to interface to the curses library. Open and Close operations are
--    provided to initialize and terminate the C program and a global
--    data structure is declared for ipc.  The global structure con-
--    tains both the timer data (elapsed time, lap num, and lap time)
--    and the command.  The update routines copy new strings to the
--    global structure and call the C program to update the display.The
--    get routine calls a C operation which gets a command and puts it
--    in the global structure.  The get routine then returns this value
--    to the caller. UPDATE_INTERVAL is initialized to the minimum,SYS-
--    TEM.TICK.  The C_TO_A_TYPE package is used to provide C compati-
--    ble data types.
-------------------------------------------------------------------------
with C_TO_A_TYPE, SYSTEM;
use C_TO_A_TYPE;
package body USER_INTERFACE is

  type TIMER_STRUCT is record    -- Ada data structure interface with C
    TIME_STR     : STRING(1 .. 12);
    LAP_TIME_STR : STRING(1 .. 12);
    LAP_NUM_STR  : STRING(1 .. 2);
    CMD          : CHAR;
  end record;
  for TIMER_STRUCT use record     -- storage allocation for the struct
    TIME_STR at 0 range 0 .. 95;
    LAP_TIME_STR at 12 range 0 .. 95;
    LAP_NUM_STR at 24 range 0 .. 15;
    CMD at 26 range 0 .. 7;
  end record;

  -- declare and link the global data structure
  type TIMER_PTR is access TIMER_STRUCT;
  TIMER : TIMER_PTR;
  pragma INTERFACE_OBJECT(TIMER, "_timer_struct");

  -- declare and link to C subprograms used by the user_interface
  procedure OPEN_UI;
  procedure CLOSE;
  procedure DISP_TIME;
  procedure GET_COMMAND;
  pragma INTERFACE (c, OPEN_UI, "_open");
  pragma INTERFACE (C, CLOSE, "_close");
  pragma INTERFACE (c, DISP_TIME, "_disp_time");
  pragma INTERFACE (c, GET_COMMAND, "_get_command");
```

```
 -- to terminate the window
 procedure CLOSE_UI is
  begin
    CLOSE;
  end CLOSE_UI;

  --put the elapsed time to global data structure and call c program
  --to update the window
  procedure UPDATE_TIME(TIME : in STRING) is
  begin
    TIMER.TIME_STR(1 .. 11) := TIME;
    DISP_TIME;
  end UPDATE_TIME;

  --put the lap time and number to global data structure and call c
  --program to update the window
  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING) is
  begin
    TIMER.LAP_TIME_STR(1 .. 11) := LAP_TIME;
    TIMER.LAP_NUM_STR(1 .. 1) := LAP_NUM;
    DISP_TIME;
  end UPDATE_LAPS;

  -- call c program to get command from the window put command to
  -- global data structure and allow this to get form the global data
  -- structure program to get
  function GET_CMD return CHARACTER is
    CH : CHARACTER := 'q';
  begin
    GET_COMMAND;
    CH := CHARACTER'VAL(TIMER.CMD);
    return CH;
  end GET_CMD;

begin
  OPEN_UI;
  UPDATE_INTERVAL := SYSTEM.TICK;
end USER_INTERFACE;
```

```
/*************************************************************************
 *  file     : ui.c
 *  author   : sun chien-hsiung
 *  description : This set of C subprograms represents a cursor-oriented
 *     user interface for the stop watch program. Those operation
 *     functions are same as the Ada using curses package operations.
 *************************************************************************/
#include <curses.h>
#include <malloc.h>
#include <stdio.h>
#include <string.h>
#include <sgtty.h>

/*
 * Declare the common data structure both the application and user
 * interface will share.
 */
struct data {
  char time[12];
  char l_time[12];
  char lap_num_str[2];
  char cmd;
};
struct data *timer_struct;

/*
 * Declare the curses window used to display the user interface.
 */
WINDOW *TIMER_WINDOW;

/*
 * Get a command from the user; put it in the global data structure.
 */
void
get_command()
{
  timer_struct->cmd = wgetch(TIMER_WINDOW);
}
/*
 * Update the window from the global data structure and refresh the
 * window.
 */
void
disp_time()
{
  mvwaddstr(TIMER_WINDOW, 1, 9, timer_struct->time);
  mvwaddstr(TIMER_WINDOW, 2, 9, timer_struct->l_time);
  mvwaddstr(TIMER_WINDOW, 2, 7, timer_struct->lap_num_str);
  wmove(TIMER_WINDOW, 3, 12);
  wrefresh(TIMER_WINDOW);
}
```

```
/*
 * Open the user interface by allocating the shared data area and
 * initializing it, creating the window and initializing it,and finally
 * calling disp to display it.
 */
void open()
{
  timer_struct = (struct data *) malloc(sizeof(struct data));
  initscr();
  clear();
  strcpy(timer_struct->time, "00:00:00:00");
  strcpy(timer_struct->l_time, "00:00:00:00");
  strcpy(timer_struct->lap_num_str, " ");
  TIMER_WINDOW = newwin(5, 23, 1, 1);
  box(TIMER_WINDOW, 'c', 'c');
  mvwaddstr(TIMER_WINDOW, 1, 2, "timer: ");
  mvwaddstr(TIMER_WINDOW, 2, 2, "lap:   ");
  mvwaddstr(TIMER_WINDOW, 3, 2, "s/r/l/q=>");
  disp_time;
}

/*
 * Close the user interface by clearing the window contents and
 * removing it from the system.
 */
void close()
{
  clear();
  refresh();
  noraw();
  echo();
  delwin(TIMER_WINDOW);
}
```

APPENDIX E

WINDOW-ORIENTED

APPLICATION-CONTROLLED USER INTERFACE

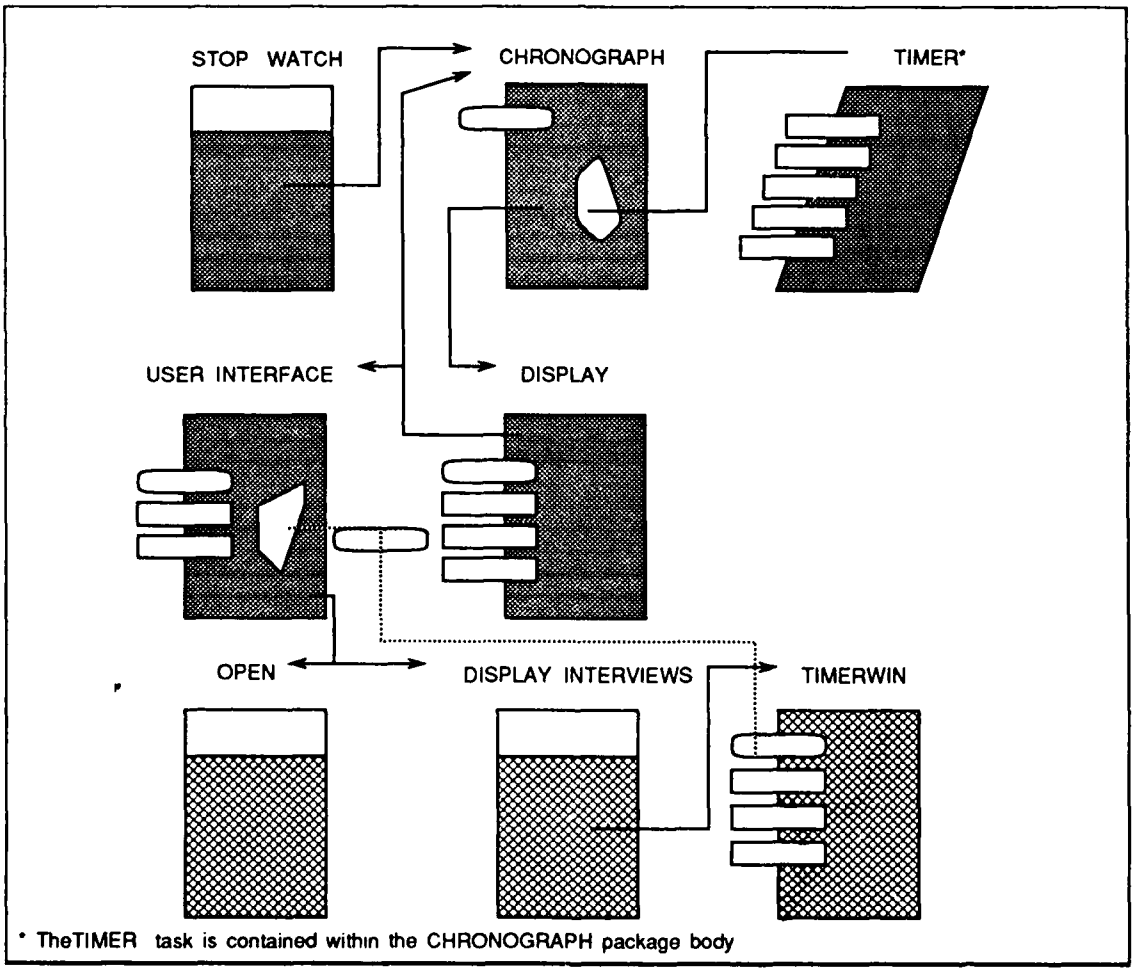

Figure E.1 Window-oriented Application-Controlled User Interface

```
-------------------------------------------------------------------------
-- file        : ui_b.a
-- author      : sun chien-hsiung
-- subject     : Window-oriented application-controlled user interface
-- description: The window-oriented Ada control the user interface.
--    Open operations are provided to initialize and close not be used.
--    The C program and a global data structure is declared for ipc.The
--    global structure contains both the timer data (elapsed time, lap
--    num, and lap time) and the command.  The update routines copy new
--    strings to the global structure. The get routine puts it in the
--    global structure.  The get routine then returns this value to the
--    caller. The C_TO_A_TYPE package is used to provide C compatible
--    data types.
-------------------------------------------------------------------------

with C_TO_A_TYPE, SYSTEM;
use  C_TO_A_TYPE;
package body USER_INTERFACE is

  type TIMER_STRUCT is record     -- Ada data structure interface with C
    TIME_STR     : STRING(1 .. 12);
    LAP_TIME_STR : STRING(1 .. 12);
    LAP_NUM_STR  : STRING(1 .. 2);
    CMD          : CHAR;
  end record;
  for TIMER_STRUCT use record     -- storage allocation for the struct
    TIME_STR at 0 range 0 .. 95;
    LAP_TIME_STR at 12 range 0 .. 95;
    LAP_NUM_STR at 24 range 0 .. 15;
    CMD at 26 range 0 .. 7;
  end record;

  -- declare and link the global data structure
  TIMER : TIMER_STRUCT;
  pragma INTERFACE_OBJECT(TIMER, "_timer_struct");

  -- declare and link to InterViews C++ subprogram used by the
  -- user_interface
  procedure GET_COMMAND;
  pragma INTERFACE(C, GET_COMMAND, "_disp_interviews__Fv");

  procedure OPEN_UI;
  pragma INTERFACE(C, OPEN_UI, "_open__Fv");

  procedure CLOSE_UI is           -- unused.
  begin
    null;
  end CLOSE_UI;
```

```
  -- simply put the new time in the global data structure.
  procedure UPDATE_TIME(TIME : in STRING) is
  begin
    TIMER.TIME_STR(1 .. 11) := TIME;
  end UPDATE_TIME;

  -- simply put the new lap num and time in the global data structure.
  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING) is
  begin
    TIMER.LAP_TIME_STR(1 .. 11) := LAP_TIME;
    TIMER.LAP_NUM_STR(1 .. 1) := LAP_NUM;
  end UPDATE_LAPS;

  -- open the user interface as a dialog box, return the current
  -- value of the global cmd when user responds.
  function GET_CMD return CHARACTER is
    CH : CHARACTER := 'q';
  begin
    GET_COMMAND;
    CH := CHARACTER'VAL(TIMER.CMD);
    return CH;
  end GET_CMD;


begin
  UPDATE_INTERVAL := SYSTEM.TICK;
  UPDATE_TIME("00:00:00:00");
  UPDATE_LAPS("00:00:00:00", "0");
  OPEN_UI;
end USER_INTERFACE;
```

```
/*************************************************************************
 * file    : tw.h
 * author : sun chien-hsiung
 * description : This is window-oriented application controlled the
 *    user interface. In this file, to define the class include opera-
 *    tions and interactors to implement the window. The MonoScene pro-
 *    vide operations for handling events, displaying interactor objects
 *    update views. The ButtonState is a class of objects that contain
 *    a value and a list of buttons that can set its value. The Message
 *    is an interactor that contains a line of text.
 *************************************************************************/

#ifndef tw_h
#define tw_h

#include <InterViews/scene.h>
#include <InterViews/button.h>
#include <InterViews/message.h>


class TimerWin:public MonoScene {
  public:
  TimerWin();
  void Run();
  virtual void Handle(Event &);
  ButtonState *press;
  Message *time_msg;
  Message *lap_time_msg;
  Message *lap_num_msg;
};

#endif
```

```
/*************************************************************************
 * file    : tw.c
 * author  : sun chien-hsiung
 * description : this file is stored TIMERWIN class operation body for
 *     the function implementation area.
 *************************************************************************/

#include "tw.h"
#include <InterViews/glue.h>
#include <InterViews/sensor.h>
#include <InterViews/button.h>
#include <InterViews/message.h>
#include <InterViews/box.h>
#include <InterViews/frame.h>

/* application - user interface global data structure */

struct data {
  char time_str[12];
  char lap_time_str[12];
  char lap_num_str[2];
  char cmd;
} timer_struct;

/* to assigned control comment to memory */

void
handle(int bp)
{
  int i;

  switch (bp) {
  case 1:
    timer_struct.cmd = 'q';/* stop */
    break;
  case 2:
    timer_struct.cmd = 's';/* start */
    break;
  case 3:
    timer_struct.cmd = 'r';/* reset */
    break;
  case 4:
    timer_struct.cmd = 'l';/* lap  */
    break;
  default:
    break;
  }
}
```

```
TimerWin: :TimerWin()
{

  /* to receive the input data from the user */
  input = new Sensor();
  input->Catch(KeyEvent);
  input->Catch(UpEvent);

  /* create the display stop watch area */

  char *time = timer_struct.time_str;
  char *lap_time = timer_struct.lap_time_str;
  char *lap_num = timer_struct.lap_num_str;

  time_msg = new Message(time);
  lap_time_msg = new Message(lap_time);
  lap_num_msg = new Message(lap_num);

  press = new ButtonState(1);

  /* building the window box */

  Insert(
    new ShadowFrame(
       new VBox(
         new VGlue(5),
         new HBox(
           new HGlue(5),
           new VBox(
             new HBox(
               new Message("timer : "),
               new HGlue(),
               time_msg
              ),
             new VGlue(2),
             new HBox(
                new Message("lap"),
                new HGlue(5),
                lap_num_msg,
                new HGlue(5),
                lap_time_msg
              ),
              new VGlue(5),
              new HBox(
                new HGlue(),
                new VBox(
                  new PushButton("Stop ", press, 1),
                  new VGlue(2),
                  new PushButton("Start", press, 2)
```

```
                ),
              new HGlue(2),
              new VBox(
                 new PushButton("Reset", press, 3),
                 new VGlue(2),
                 new PushButton(" Lap ", press, 4)
               ),
               new HGlue()
              )
             ),
             new HGlue(5)
           ),
           new VGlue(5)
         )
       )
     );
}

/*
 * It's a loop for detect the input event, their are keyboard or button,
 * to decision the program to call handle to update the command or
 * terminate the program.
 */
void TimerWin::Run()
{
  Event e;
  int v;
  do {
    Read(e);
    e.target->Handle(e);
    if (e.eventType == UpEvent) {
      press->GetValue(v);
      handle(v);
      press->SetValue(0);
      e.target = nil;
    }
  } while (e.target != nil);
}

/*
 * This is handle the keyboard event it is 'q' than terminal the program
 */
void TimerWin::Handle(Event & e)
{
  if (e.eventType == KeyEvent)
    if (e.len > 0 && e.keystring[0] == 'q') {
      timer_struct.cmd = 'k';
      e.target = nil;
    }
}
```

```
/*************************************************************************
 * file   : main.c
 * author : sun chien hsiung
 * description : This is main program for user interface. Here, create
 *    the world which is a application's root scene. An instance of
 *    TIMERWIN is created and inserted to the world.
 *************************************************************************/
#include "tw.h"
#include <InterViews/world.h>
#include <stdio.h>

/*
 * define font for output text line
 */
static PropertyData properties[] = {
  {"*Message*font", "*-courier-bold-r-*-140-*"},
  {"*PushButton*font", "*-courier-bold-r-*-100-*"},
  {nil}
};

static OptionDesc options[] = {
  {nil}
};

World *world;

/*
 * define/create the World for implement window
 */
void open()
{
  int argc = 0;
  char **argv;
  char *name = "open_tw";
  argv = &name;
  world = new World("TimerWin", properties, options, argc, argv);
}
/*
 * using the popup window to display the SW
 */
void disp_interviews()
{
  TimerWin *t = new TimerWin();

  t->SetName("StopWatch");
  t->SetIconName("StopWatch");
  world->InsertPopup(t);
  t->Run();
  world->Remove(t);
  delete t;
}
```

APPENDIX F

WINDOW-ORIENTED

USER INTERFACE-CONTROLLED APPLICATION

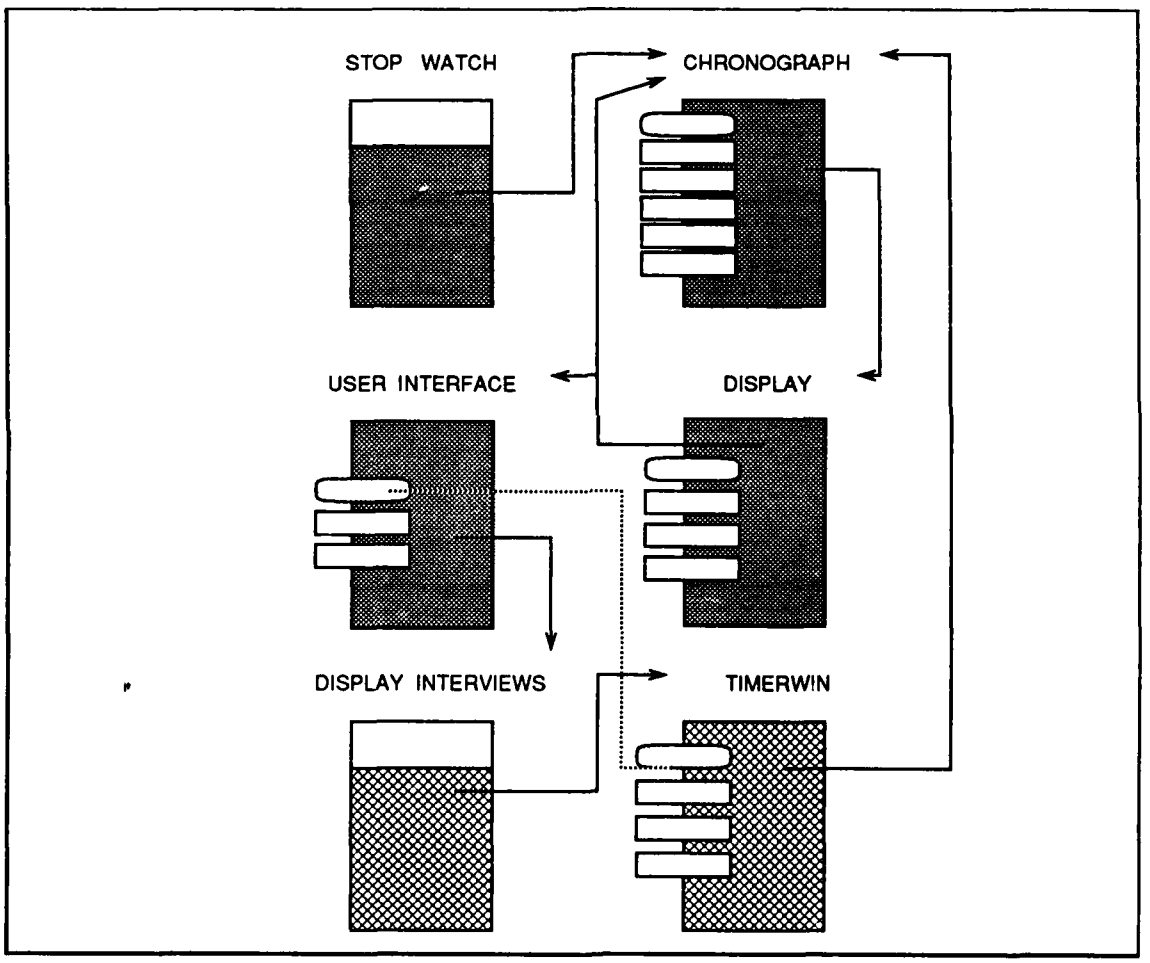

Figure F.1 Window-oriented User Interface-Controlled Application

```
-------------------------------------------------------------------------
-- file        : chron_s.a
-- author      : sun chien-hsiung
-- subject     : window-oriented user interface in charge.
-- description: The chronograph package defines the abstract state ma-
--    chine used in the stop watch program.  Each states function that
--    implement by it own procedure Rather than using the task entry
--    call. Those procedure will be call by directly from the user
--    interface.
-------------------------------------------------------------------------

package CHRONOGRAPH is

  subtype NUM_LAPS   is INTEGER range 0 .. 9;
  subtype TWO_DIGITS is INTEGER range 0 .. 99;

  -- Timer_Record represents the time values to be exported to using
  -- programs
  type TIMER_RECORD is record
    HOURS       : TWO_DIGITS := 0;
    MINUTES     : TWO_DIGITS := 0;
    SECONDS     : TWO_DIGITS := 0;
    HUNDREDTHS : TWO_DIGITS := 0;
  end record;

  type COMMAND is (START, STOP, LAP, RESET, KILL);

  -- operations will be called by the user interface
  procedure START;
  procedure STOP;
  procedure LAP_TIMER;
  procedure RESET_TIMER;
  procedure UPDATE;
  pragma EXTERNAL_NAME(START, "_start__Fv");
  pragma EXTERNAL_NAME(STOP, "_stop__Fv");
  pragma EXTERNAL_NAME(RESET_TIMER, "_reset__Fv");
  pragma EXTERNAL_NAME(LAP_TIMER, "_lap__Fv");
  pragma EXTERNAL_NAME(UPDATE, "_update_data__Fv");

end CHRONOGRAPH;
```

```
-------------------------------------------------------------------------
-- file         : chron_b.a
-- author       : sun chien-hsiung
-- subject      : An abstract state machine to perform stop watch
--                functions.
-- Description : The chronograph contains four operations, internal
--    elapsed time, lap times, and the timer state.  Operations are
--    start, stop, reset, and record/review laps, which rather than in
--    the task for function entry call, their are called by the user
--    interface process.
-------------------------------------------------------------------------
with CALENDAR, DISPLAY;
use  CALENDAR;
package body CHRONOGRAPH is

  type LAP_ARRAY is array(NUM_LAPS) of DURATION;
  type STATES    is (INITL, RUNNING, STOPPED);
  ELTIME             : DURATION     := 0.0;
  DISP_LAP           : NUM_LAPS     := NUM_LAPS'FIRST;
  LAPS               : LAP_ARRAY    := (others => 0.0);
  NEXT_LAP           : NUM_LAPS     := NUM_LAPS'FIRST;
  STATE              : STATES       := INITL;
  START_TIME         : TIME         := CLOCK;
  EL_TIME, LAP_TIME  : TIMER_RECORD := (others => 0);

  -- convert system time to two digit numbers
  function CONVERT_TIME(TIME : in DURATION) return TIMER_RECORD is
    INT_TIME   : INTEGER;
    TIMER_TIME : TIMER_RECORD;
  begin
    INT_TIME := INTEGER(TIME);
    TIMER_TIME.HOURS := INT_TIME / 3600;
    TIMER_TIME.MINUTES := INT_TIME mod 3600 / 60;
    TIMER_TIME.SECONDS := INT_TIME mod 60;
    TIMER_TIME.HUNDREDTHS := INTEGER(FLOAT(TIME) * 100.0) mod 100;
    return TIMER_TIME;
  end CONVERT_TIME;

  -- when the command is input start to start running timer
  procedure START is
  begin
    case STATE is
      when INITL =>
        START_TIME := CLOCK;
      when STOPPED =>
        START_TIME := CLOCK - ELTIME;
      when others =>
        null;
    end case;
```

```
  STATE := RUNNING;
  EL_TIME := CONVERT_TIME(ELTIME);
  DISPLAY.PUT(EL_TIME);
end START;

-- when the command is input stop than stop the timer
procedure STOP is
begin
  case STATE is
    when RUNNING =>
      ELTIME := CLOCK - START_TIME;
      STATE := STOPPED;
    when others =>
      null;
  end case;
  EL_TIME := CONVERT_TIME(ELTIME);
  DISPLAY.PUT(EL_TIME);
end STOP;

-- when the command is input lap than record the current
procedure LAP is
begin
  case STATE is
    when RUNNING =>
      if NEXT_LAP <= NUM_LAPS'LAST then
        DISP_LAP := NEXT_LAP;
        LAPS(DISP_LAP) := CLOCK - START_TIME;
        if NEXT_LAP < NUM_LAPS'LAST then
          NEXT_LAP := NUM_LAPS'SUCC(DISP_LAP);
        end if;
      end if;
    when STOPPED =>
      if NEXT_LAP > NUM_LAPS'FIRST and then DISP_LAP <
       NUM_LAPS'PRED(NEXT_LAP) then
        DISP_LAP := NUM_LAPS'SUCC(DISP_LAP);
      else
        DISP_LAP := NUM_LAPS'FIRST;
      end if;
    when others =>
      null;
  end case;
  LAP_TIME := CONVERT_TIME(LAPS(DISP_LAP));
  DISPLAY.PUT(LAP_TIME, DISP_LAP);
end LAP;
```

```
    -- when the command is input reset than to initialized the timer
    procedure RESET is
    begin
      case STATE is
        when STOPPED | RUNNING =>
          NEXT_LAP := LAP_ARRAY'FIRST;
          DISP_LAP := LAP_ARRAY'FIRST;
          LAPS := (others => 0.0);
          STATE := INITL;
          ELTIME := 0.0;
        when others =>
          null;
      end case;
      EL_TIME := CONVERT_TIME(ELTIME);
      LAP_TIME := CONVERT_TIME(LAPS(DISP_LAP));
      DISPLAY.PUT(EL_TIME);
      DISPLAY.PUT(LAP_TIME, DISP_LAP);
    end RESET;

    -- when user interface needed to update the display time call
    -- application to update
    procedure UPDATE is
    begin
      if STATE = RUNNING then
        ELTIME := CLOCK - START_TIME;
        EL_TIME := CONVERT_TIME(ELTIME);
        DISPLAY.PUT(EL_TIME);
      end if;
    end UPDATE;

  end CHRONOGRAPH;
```

```
-------------------------------------------------------------------------
-- file        : disp_s.a
-- author      : sun chien-hsiung
-- subject     : This package represents the system display device.
-- description: Put operation is provided for updating time and
--     lap time/numbers..
-------------------------------------------------------------------------
with CHRONOGRAPH;
use  CHRONOGRAPH;

package DISPLAY is

  procedure PUT(EL_TIME : in TIMER_RECORD);
  procedure PUT(LAP_TIME : in TIMER_RECORD; LAP : in NUM_LAPS);

end DISPLAY;
```

```
-------------------------------------------------------------------------
-- file        : disp_b.a
-- author      : sun chien-hsiung
-- subject     : implements I/O for the Stop Watch program.
-- description: I/O is limited to implementation hidden in user
--    interface package. On output, values received from the Timer ASM
--    are converted to strings and the UI package routines are called.
--    Likewise, commands are retrieved in character form and converted
--    to the enumerated type specified by the CHRONOGRAPH package.
-------------------------------------------------------------------------
with USER_INTERFACE;
package body DISPLAY is

-- convert the integer time to string

  function INT_TO_STRING(NUMBER, WIDTH : in INTEGER) return STRING is
    NEW_STRING : STRING(1 .. WIDTH) := (others => '0');
    DIGIT      : INTEGER;
    NEW_NUMBER : INTEGER             := NUMBER;
  begin
    for I in reverse 1 .. WIDTH loop
      DIGIT := NEW_NUMBER mod 10;
      NEW_STRING(I) := CHARACTER'VAL(DIGIT + 48);
      NEW_NUMBER := NEW_NUMBER / 10;
    end loop;
    return NEW_STRING;
  end INT_TO_STRING;

  -- put times(hh,mm,ss,ms) to as a display string

   function TIME_TO_STRING(TIME : in TIMER_RECORD) return STRING is
    NEW_STRING : STRING(1 .. 11) := "00:00:00:00";
  begin
    NEW_STRING(1 .. 2)  := INT_TO_STRING(TIME.HOURS, 2);
    NEW_STRING(4 .. 5)  := INT_TO_STRING(TIME.MINUTES, 2);
    NEW_STRING(7 .. 8)  := INT_TO_STRING(TIME.SECONDS, 2);
    NEW_STRING(10 .. 11) := INT_TO_STRING(TIME.HUNDREDTHS, 2);
    return NEW_STRING;
  end TIME_TO_STRING;

  -- put display time string to User_Interface.

   procedure PUT(EL_TIME : in TIMER_RECORD) is
  begin
    USER_INTERFACE.UPDATE_TIME(TIME_TO_STRING(EL_TIME));
  end PUT;
```

```
   -- put lap time string

  procedure PUT(LAP_TIME : in TIMER_RECORD; LAP : in NUM_LAPS) is
  begin
    USER_INTERFACE.UPDATE_LAPS(TIME_TO_STRING(LAP_TIME),
                               INT_TO_STRING(LAP,1));
  end PUT;

end DISPLAY;
```

```
-------------------------------------------------------------------
-- file        : ui_s.a
-- author      : sun chien-hsiung
-- subject     : User interface implementation for Stop Watch program.
-- description: The user interface provides implementation specific
--    details depending on style of user interface.  The specification,
--    however, does not change, only the body.  Various types of user
--    interfaces include:
--    line-oriented
--    cursor-oriented
--    window-oriented
-- changes     : window-oriented user-interface-controlled:  eliminate
--    update_interval, close_ui, and get_cmd.
-------------------------------------------------------------------

with SYSTEM;
package USER_INTERFACE is
  procedure UPDATE_TIME(TIME : in STRING);
  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING);
end USER_INTERFACE;
```

```
-------------------------------------------------------------------------
-- file        : ui_b.a
-- author      : sun chien-hsiung
-- subject     : window-oriented user interface control application for
--               Stop Watch
-- description : The user interface in charge, Ada need to call the user
--     interface,after that, control by the user interface. In this pro-
--     gram, the close is not used and update operations put the update
--     timer to the global data structure.
-------------------------------------------------------------------------
with C_TO_A_TYPE, CHRONOGRAPH, SYSTEM;
use  C_TO_A_TYPE, CHRONOGRAPH;
pragma ELABORATE(CHRONOGRAPH);
package body USER_INTERFACE is

  type TIMER_STRUCT is record    -- Ada data structure interface with C
    TIME_STR     : STRING(1 .. 12);
    LAP_TIME_STR : STRING(1 .. 12);
    LAP_NUM_STR  : STRING(1 .. 2);
    CMD          : CHAR;
    TIME_CHG     : CHAR;
    LAP_CHG      : CHAR;
  end record;
  for TIMER_STRUCT use record    -- storage allocation for the struct
    TIME_STR at 0 range 0 .. 95;
    LAP_TIME_STR at 12 range 0 .. 95;
    LAP_NUM_STR at 24 range 0 .. 15;
    CMD at 26 range 0 .. 7;
    TIME_CHG at 27 range 0 .. 7;
    LAP_CHG at 28 range 0 .. 7;
  end record;

  -- declare and link the global data structure
  TIMER : TIMER_STRUCT;
  pragma INTERFACE_OBJECT(TIMER, "_timer_struct");

  -- declare and link to InterViews C++ subprogram used by the
  -- user_interface
  procedure OPEN_UI;
  pragma INTERFACE(C, OPEN_UI, "_disp_interviews__Fv");
  procedure CLOSE_UI is          -- unused.
  begin
    null;
  end CLOSE_UI;

  -- simply put the new time in the global data structure.
  procedure UPDATE_TIME(TIME : in STRING) is
  begin
    TIMER.TIME_STR(1 .. 11) := TIME;
    TIMER.TIME_CHG := CHARACTER'POS('Y');
  end UPDATE_TIME;
```

```
    -- simply put the new lap num and time in the global data structure.
    procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING) is
    begin
      TIMER.LAP_TIME_STR(1 .. 11) := LAP_TIME;
      TIMER.LAP_NUM_STR(1 .. 1) := LAP_NUM;
      TIMER.LAP_CHG := CHARACTER'POS('Y');
    end UPDATE_LAPS;

  begin
    TIMER.TIME_CHG := CHARACTER'POS('Y');
    TIMER.LAP_CHG := CHARACTER'POS('Y');
    UPDATE_TIME("00:00:00:00");
    UPDATE_LAPS("00:00:00:00", "0");
    OPEN_UI;
  end USER_INTERFACE;
```

```
/*************************************************************************
 * file        : tw.c
 * author      : sun chien hsiung
 * description : The window-oriented user interface-controlled
 *               application
 *     This version calls application subroutines to execute functional-
 *     ity. Handle directly call start, stop, reset, and lap procedures
 *     in the Ada program in response to user button selections.  Up-
date
 *     is called to perform the stop watch timer update and redraw the
 *     applicable message.  Update is also called in response to timer
 *     events.
 *************************************************************************/

#include "tw.h"
#include <InterViews/glue.h>
#include <InterViews/sensor.h>
#include <InterViews/button.h>
#include <InterViews/message.h>
#include <InterViews/box.h>
#include <InterViews/Std/string.h>
#include <stdio.h>

/* shared memory data structure */

struct data {
   char time_str[12];
   char lap_time_str[12];
   char lap_num_str[2];
 char cmd;
   char time_chg;
   char lap_chg;
} timer_struct;

extern void start();
extern void stop();
extern void reset();
extern void lap();
extern void update();

/* to assigned control comment to memory */

void handle (int bp) {
  int i;
  switch (bp) {
    case 1:
      stop();   /* stop */
      break;
```

```
    case 2:
      start();   /* start */
      break;
    case 3:
      reset();   /* reset */
      break;
    case 4:
      lap();   /* lap  */
      break;
    default:
      break;
  }
}

TimerWin::TimerWin (int delay){

/*  to receive the input data from the user */
  input = new Sensor();
  input->Catch(KeyEvent);
  input->Catch(UpEvent);
  input->CatchTimer(delay,1);

/*  creat the display stop watch area */

  char* time = timer_struct.time_str;
  char* lap_time = timer_struct.lap_time_str;
  char* lap_num = timer_struct.lap_num_str;
  time_msg = new Message(time);
  lap_time_msg = new Message(lap_time);
  lap_num_msg = new Message(lap_num);

  press = new ButtonState(1);

/* building the window box */

  Insert(
    new VBox(
      new VGlue(5),
      new HBox(
        new HGlue(5),
        new VBox(
          new HBox(
            new Message ("timer : "),
            new HGlue(),
            time_msg
          ),
          new VGlue(2),
          new HBox(
            new Message ("lap"),
            new HGlue(5),
            lap_num_msg,
```

```
            new HGlue(5),
            lap_time_msg
          ),
          new VGlue(5),
          new HBox(
            new HGlue(),
            new VBox(
              new PushButton("Stop ", press, 1),
              new VGlue(2),
              new PushButton("Start", press, 2)
            ),
            new HGlue(2),
            new VBox(
              new PushButton("Reset", press, 3),
              new VGlue(2),
              new PushButton(" Lap ", press, 4)
            ),
            new HGlue()
          )
        ),
        new HGlue(5)
      ),
      new VGlue(5)
    )
  );
}

void TimerWin::Run () {
  Event e;
  int v;
  do {
    Read(e);
    e.target->Handle(e);
    if (e.eventType == UpEvent) {
      press->GetValue(v);
      handle(v);
      press->SetValue(0);
      Update();
    }
    if (e.eventType == TimerEvent) {
      update();
      Update();
    }
  } while (e.target != nil);
}
void TimerWin::Handle (Event& e) {
  if (e.eventType == KeyEvent)
    if (e.len > 0 && e.keystring[0] == 'q'){
      e.target = nil;
    }
}
```

```
void TimerWin::Update() {
  if (timer_struct.time_chg == 'y')  {
    time_msg->Draw();
    timer_struct.time_chg = 'n';
  }
  if (timer_struct.lap_chg == 'y') {
    lap_time_msg->Draw();
    lap_num_msg->Draw();
    timer_struct.lap_chg = 'n';
  }
}
```

```
/*************************************************************************
 * file   : main.c
 * author : sun chien hsiung
 * description : This program is main program for the user interface
 *     using the window to display the timer stop watch.
 *************************************************************************/
#include "tw.h"
#include <InterViews/world.h>
#include <InterViews/frame.h>
#include <stdio.h>
/*
 * define font for output text line
 */
static PropertyData properties[] = {
  {"*Message*font", "*-courier-bold-r-*-140-*"},
  {"*PushButton*font", "*-courier-bold-r-*-100-*"},
  {"*delay", "0.1"},
  {nil}
};

static OptionDesc options[] = {
  {"-delay", "*delay", OptionValueNext},
  {nil}
};

void disp_interviews()
{
  int argc = 0;
  char **argv;
  char *name = "open_tw";
  argv = &name;
  World *world = new World("TimerWin", properties, options, argc, argv);
  int delay = atoi(world->GetAttribute("delay"));
  TimerWin *t = new TimerWin(delay);
  t->SetName("* STOP WATCH *");
  t->SetIconName("ISW");
  world->InsertApplication(t);
  t->Run();
}
```

APPENDIX G

WINDOW-ORIENTED

ASYNCHRONOUS-CONTROLLED

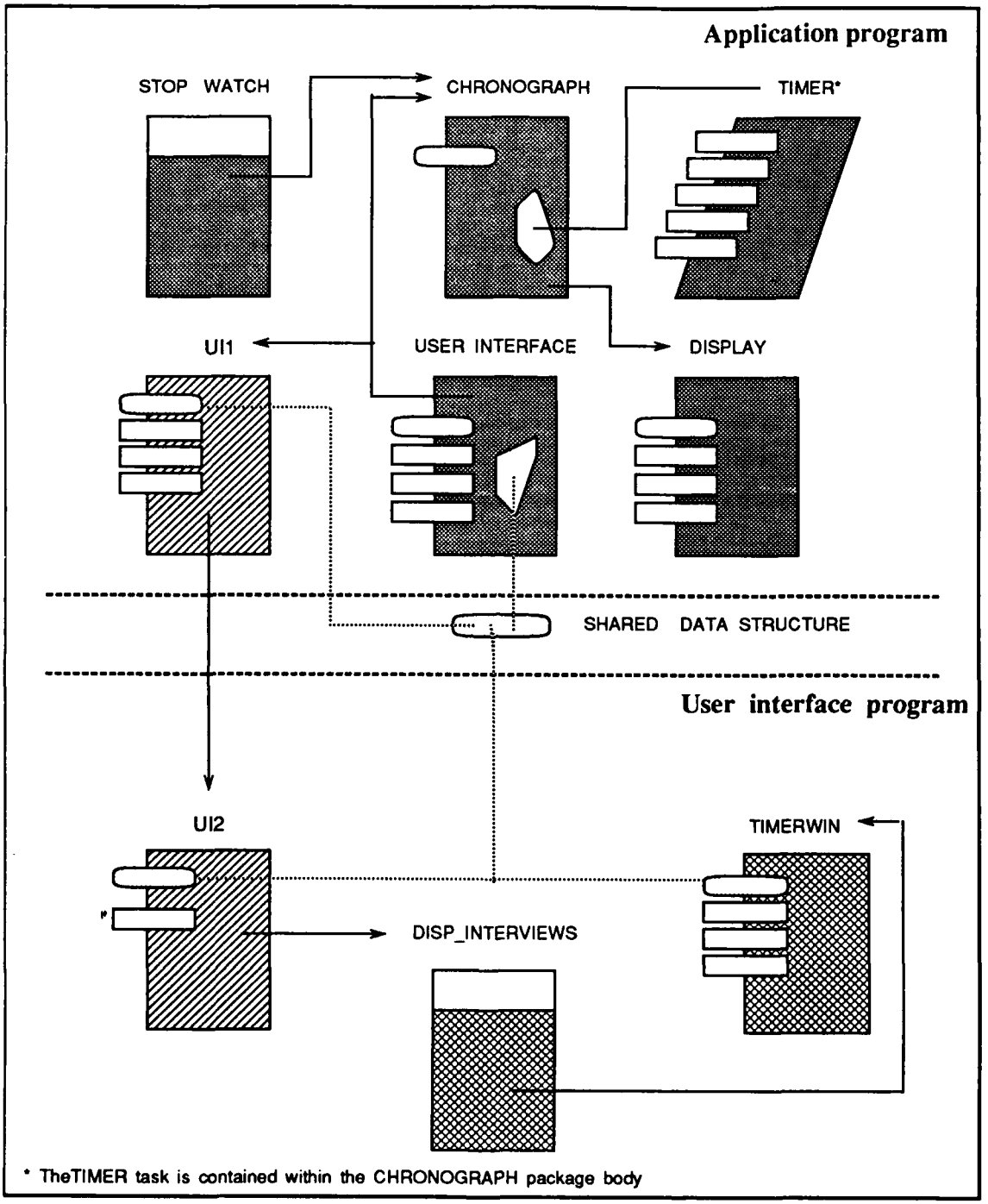

Figure G.1 Window-oriented User Interface-Controlled Application

```
-------------------------------------------------------------------------
--  file        : ui_b.a
--- author      : sun chien-hsiung
-- subject     : Window-oriented using asynchronous process for Stop
--               Watch
-- description: The asynchronous process is needed using the system
--      command to control the shared memory area. In this program, that
--      uses three operations to interface c program which is using the
--      perform the shared memory open/attach, fork the interface
--      process, and close the shared memory. All data communication
--      only use the shared global data area.
-------------------------------------------------------------------------

with C_TO_A_TYPE, SYSTEM;
use  C_TO_A_TYPE;
package body USER_INTERFACE is

  type TIMER_STRUCT is record     -- Ada data structure interface with C
    TIME_STR     : STRING(1 .. 12);
    LAP_TIME_STR : STRING(1 .. 12);
    LAP_NUM_STR  : STRING(1 .. 2);
    CMD          : CHAR;
    TIME_CHG     : CHAR;
    LAP_CHG      : CHAR;
  end record;
  for TIMER_STRUCT use record     -- storage allocation for the struct
    TIME_STR at 0 range 0 .. 95;
    LAP_TIME_STR at 12 range 0 .. 95;
    LAP_NUM_STR at 24 range 0 .. 15;
    CMD at 26 range 0 .. 7;
    TIME_CHG at 27 range 0 .. 7;
    LAP_CHG at 28 range 0 .. 7;
  end record;

  -- declare and link the global data structure
  type TIMER_PTR is access TIMER_STRUCT;
  TIMER : TIMER_PTR;
  pragma INTERFACE_OBJECT(TIMER, "_timer_struct");

  -- declare and link to C subprograms used by the user_interface
  procedure OPEN_UI;              -- open shared memory
  procedure CLOSE;                -- close the shared memory
  procedure SPAWN_UI;             -- fork user interface process
  pragma INTERFACE(C, OPEN_UI, "_open_share_memory");
  pragma INTERFACE(C, CLOSE, "_close_share_memory");
  pragma INTERFACE(C, SPAWN_UI, "_spawn_UI");
```

```
  -- to close the shared memory
  procedure CLOSE_UI is
  begin
    CLOSE;
  end CLOSE_UI;

  -- put the update time to the global data structure and assign the
  -- flag to 'y' which is allow window to update
  procedure UPDATE_TIME(TIME : in STRING) is
  begin
    TIMER.TIME_STR(1 .. 11) := TIME;
    TIMER.TIME_CHG := CHARACTER'POS('Y');
  end UPDATE_TIME;

  -- put the update lap time tnd number to the global data structure and
  -- assign the flag to 'y' which is allow window to update
  procedure UPDATE_LAPS(LAP_TIME : in STRING; LAP_NUM : in STRING) is
  begin
    TIMER.LAP_TIME_STR(1 .. 11) := LAP_TIME;
    TIMER.LAP_NUM_STR(1 .. 1) := LAP_NUM;
    TIMER.LAP_CHG := CHARACTER'POS('Y');
  end UPDATE_LAPS;

  -- using the  semaphore  concept to control between the application
  -- and user interface to use the shared memory command. Here. Using
  -- while to control.
  function GET_CMD return CHARACTER is
    CH : CHARACTER := 'q';
  begin
    CH := CHARACTER'VAL(TIMER.CMD);
    while CHARACTER'VAL(TIMER.CMD) = ' ' loop
      null;
    end loop;
    return CH;
  end GET_CMD;

begin
  OPEN_UI;
  SPAWN_UI;
  UPDATE_INTERVAL := SYSTEM.TICK;
end USER_INTERFACE;
```

```
/*************************************************************************
 * file     :uil.c
 * author   : sun chien hsiung
 * description : the purpose is allows Ada program to communicate user
 *   interface as separate UNIX process thorough a shared data struc-
 *   ture.This file including three program the fist one is open
 *   shared memory,the second is fork to interface process, and the
 *   last one is to close the shared memory.
 *************************************************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>
#include <errno.h>

/* shared memory data structure */

struct data {
  char time_str[12];
  char lap_time_str[12];
  char lap_num_str[2];
  char cmd;
  char time_chg;
  char lap_chg;
};
struct data *p;

key_tmemory = 123456L;              /* address for the memory */
int mid;                             /* get memory ID          */
int shmflag = IPC_CREAT|IPC_EXCL|0666;

open_share_memory(){
int i;
/*
 * create a shared memory segment for the user interface
 */

   mid = shmget(memory,sizeof(struct data),shmflag);
   if (mid < 0) {
   mid = shmget(memory,sizeof(struct data),0200);
  }

/*
 * attach process
 */

   p = (struct data *) shmat(mid,(char *) 0, 0);
   if (p == (struct data *) -1)
   perror("server:shmat");

}
```

```
/* spawn the user interface */

spawn_UI(){
  if (fork() == 0) {
    execlp("ui","ui",NULL);
    perror("fork of user interface failed");
    exit(-1);
  }
}

  /* remove the shared memory segment */

void close_share_memory(){
  mid = shmget(memory,sizeof(struct data),0200);
    if (shmctl(mid,IPC_RMID) == -1)
      perror("server:shmctl(remove)");
  exit(0);
}
```

```
/*************************************************************************
 * file  : ui2.c
 * author : sun chien hsiung
 * description : This purpose of this program is separate shared memory
 *     segment from user interface program, fork by the application pro-
 *     gram and call the Interviews program.
 *************************************************************************/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <signal.h>

struct data {
  char time[12];
  char l_time[12];
  char lap_num_str[2];
  char cmd;
  char time_chg;
  char lap_chg;
};

struct data *p;
key_t memory = 123456L;
int mid;
int shmflag = 0666;
int was_working = 0;

extern void disp_interviews();
void main()
{

 /*
 * create a shared memory segment for the user interface
 */
       *
mid = shmget(memory, sizeof(struct data), shmflag);
  if (mid < 0) {
    perror("client:shmget");
  }

/*
 * attach our process to it
 */

  p = (struct data *) shmat(mid, (char *) 0, 0);
  if (p == (struct data *) - 1)
    perror("client:shmat");
  disp_interviews();
}
```

```
/*************************************************************************
 * file    : tw.c
 * author  : sun chien hsiung
 * description : The window-oriented asynchronous control application
 *     This is program needed to call application subroutine to
 *     implement functions. In this program, is combimnation of above
 *     two control program to perform the window
*************************************************************************/
/*
 * TimerWin class
 */
#include "tw.h"
#include <InterViews/glue.h>
#include <InterViews/sensor.h>
#include <InterViews/button.h>
#include <InterViews/message.h>
#include <InterViews/box.h>
#include <InterViews/Std/string.h>
#include <stdio.h>

/* shared memory data structure */
struct data {
   char time_str[12];
   char lap_time_str[12];
   char lap_num_str[2];
   char cmd;
   char time_chg;
   char lap_chg;
};

extern struct data *p;

/* to assigned control comment to memory */

void handle (int bp) {
  int i;
  while (p->cmd != ' ') {};
  switch (bp) {
    case 1:
      p->cmd =  'Q';   /* stop */
      break;
    case 2:
      p->cmd =  'S';   /* start */
      break;
    case 3:
      p->cmd =  'R';   /* reset */
      break;
```

```
    case 4:
      p->cmd =  'L';    /* lap  */
      break;
    default:
      break;
  }
}
TimerWin::TimerWin (int delay){
/*  to receive the input data from the user */
  input = new Sensor();
  input->Catch(KeyEvent);
  input->Catch(UpEvent);
  input->CatchTimer(delay,1);

/*  creat the display stop watch area */
  char* time = p->time_str;
  char* lap_time = p->lap_time_str;
  char* lap_num = p->lap_num_str;
  time_msg = new Message(time);
  lap_time_msg = new Message(lap_time);
  lap_num_msg = new Message(lap_num);
  press = new ButtonState(1);

/* building the window box */
  Insert(
    new VBox(
      new VGlue(5),
      new HBox(
        new HGlue(5),
        new VBox(
          new HBox(
            new Message ("timer : "),
            new HGlue(),
            time_msg
          ),
          new VGlue(2),
          new HBox(
            new Message ("lap"),
            new HGlue(5),
            lap_num_msg,
            new HGlue(5),
            lap_time_msg
          ),
          new VGlue(5),
          new HBox(
            new HGlue(),
            new VBox(
              new PushButton("Stop ", press, 1),
              new VGlue(2),
              new PushButton("Start", press, 2)
            ),
```

```
            new HGlue(2),
            new VBox(
              new PushButton("Reset", press, 3),
              new VGlue(2),
              new PushButton(" Lap ", press, 4)
            ),
            new HGlue()
          )
        ),
        new HGlue(5)
      ),
      new VGlue(5)
    )
  );
}
/*
 * It's a loop for detect the input event, their are keyboard or button,
 * to decision the program to call handle to update the command or
 * terminate the program.
 */
void TimerWin::Run () {
  Event e;
  int v;
  do {
    Read(e);
    e.target->Handle(e);
    if (e.eventType == UpEvent) {
      press->GetValue(v);
      handle(v);
      Update();
      press->SetValue(7);
    }
    if (e.eventType == TimerEvent) {
      Update();
    }
  } while (e.target != nil);
}
/*
 * This is handle the keyboard event it is 'q' than terminal the program
 */
void TimerWin::Handle (Event& e) {
  if (e.eventType == KeyEvent)
    if (e.len > 0 && e.keystring[0] == 'q'){
      p->cmd = 'K';
      e.target = nil;
    }
}
```

```
/*
 * update the window elapsed time, lap time, and lap number
 */
void TimerWin::Update() {
  if (p->time_chg == 'y')  {
    time_msg->Draw();
    p->time_chg = 'n';
  }
  if (p->lap_chg == 'y') {
    lap_time_msg->Draw();
    lap_num_msg->Draw();
    p->lap_chg = 'n';
  }
 }
```

APPENDIX H

BITMAPS EXECUTOR PGROGRAM CODE

```
/* This is listing the bmx program main different  than the Logo program */
#include "bmx.h"
#include <InterViews/bitmap.h>
#include <InterViews/painter.h>
#include <InterViews/pattern.h>
#include <InterViews/sensor.h>
#include <InterViews/shape.h>
#include <InterViews/transformer.h>
#include <InterViews/color.h>
#include <InterViews/Std/math.h>
#include <InterViews/Std/stdio.h>
#include <InterViews/Std/string.h>
#include <InterViews/Std/stdlib.h>

char pgm_str[80] = "xterm +sb ";

BitmapExec::BitmapExec () {
   SetClassName("BitmapExec");
   input = new Sensor();
   input->Catch(KeyEvent);
   input->Catch(UpEvent);
   bitmap = nil;
   rainbow = 0;
   const char* options = GetAttribute("xtermopts");
   if (options != nil)
     strcat(pgm_str, options);
   const char* name = GetAttribute("program");
   if (name != nil) {
     strcat(pgm_str, " -e ");
     strcat(pgm_str, name);
     strcat(pgm_str, " &");
   }
}
```

```
void BitmapExec::Handle (Event& e) {
   if (e.eventType == KeyEvent && e.keystring[0] == 'q') {
      e.target = nil;
   } else if (e.eventType == UpEvent) {
      const char* name = GetAttribute("program");
      system(pgm_str);
   }
}
static OptionDesc options[] = {
   { "-r", "*rainbow", OptionValueNext },
   { "-b", "*bitmap", OptionValueNext },
   { "-o", "*xtermopts", OptionValueNext },
   { "-p", "*program", OptionValueNext },
   { nil }
};
```

LIST OF REFERENCES

1. Commander, Naval Sea Systems Command UNCLASSIFIED Letter 9410 OPR:61Y Serial 61Y/1036 to Superintendent, Naval Postgraduate School, *Subject: Statement of Work for Low Cost Combat Direction System*, 20 December 1988.
2. Jones, O., *Introduction To The X Window System*, Prentice Hall, 1988.
3. D.O.D, *Reference mannul for the Ada Programming Language*, ANSI/MIC-STD-1815A-1983, American National Standards Institute, 1983.
4. Department of the Navy, (NAVSEA) 0967-LP-027-8602, *SystemsEngineeriHandbook* Vol. I, Combat Direction System Model 5, February 1985.
5. Lanrel. M., *Applied Physics Laboratory*, The Johns Hopkins Universsity.
6. Naval Research Advisory Committee, *Next Generation Computer Resources*, February 1989.
7. Szekely, P. and Myers, B., "A User Interface Toolkit Based on Graphical on Graphical Objects and Constraints", in *Proceedings of ACM OOPSLA 88 Conference*, September 1988.
8. Booth, G., *Software Engineering With Ada*, 2d ed., Binjamin/Cummings Publishing Co., Menlo Park, 1987.
9. Barnes, J. G. P., *Programming In Ada*, 3d ed., Addison Wesley, 1989.
10. Luqi, *Computer Aided Maintenance of Prototype Systems*, Technical Report NPS 52-88-037, Computer Science Department, Naval Postgraduate School, Monterey, CA, September 1988.
11. Shumate, K., *Understanding Concurrincy in Ada*, Intertext/Multiscience/McGraw Hall, 1988.
12. Skansholm, J., *Ada From The Beginning*, Addison Wesley, 1988.
13. Quercia, V. and O'Reilly, T., *The Definitive Guides to The X Window System :X Window System User's Guide*, vol2, v 11, O'Reilly & Associates, 1988.
14. Scheifler, R. W./ Gettys, T. and Newman, R., *X Window System C Library and Protocol reference*, Digital press, 1988.
15. Linton, M. A./Vlissides, J. M./Calder, P.R., *Composing User Interfaces with InterViews*, Stanford University.

16. Strang, J., *A Nutshell Handbook for Progrmming with Curses*, O'Reilly &Assosiates, 1986.

17. Jim, A. S. and Guenter P. S., *Requirements analysis for Low Cost Combat Direction System*, Masters thisis, US Naval Postgraduate School, Monterey Ca., December 1990.

18. VERDIX Ada Development, *VADS Version 5.5 for SUN-3*, VERDX Corpotation, 1988 .

19. Sun Microsystems, *UNIX User's Reference Mannal More/bsd Version*, Sun Microsystems Inc, Berkeley, 1986.

20. Mark, L., *InterViews mannul Version 2.5* , Stanford University, 1987.

21. Massachusetts Institute of Technology, *X Window System Version 11 release 3*,1988.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Director of Research Administration Code 012 Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Computer Science Department Naval Postgraduate School Monterey, CA 93943	1
Office of Naval Research 800 N. Quincy Street Arlington, VA 22217-5000	1
Center for Naval Analysis 4401 Ford Avenue Alexandria, VA 22302-0268	1
National Science Foundation Division of Computer and Computation Research Washington, D.C. 20550	1
Office of the Chief of Naval Operations Code OP-941 Washington, D.C. 20350	1
Office of the Chief of Naval Operations Code OP-945 Washington, D.C. 20350	1

Commander Naval Telecommunications Command 2
Naval Telecommunications Command Headquarters
4401 Massachusetts Avenue NW
Washington, D.C. 20390-5290

Commander Naval Data Automation Command 1
Washington Navy Yard
Washington, D.C. 20374-1662

Dr. Lui Sha 1
Carnegie Mellon University
Software Engineering Institute
Department of Computer Science
Pittsburgh, PA 15260

COL C. Cox, USAF 1
JCS (J-8)
Nuclear Force Analysis Division
Pentagon
Washington, D.C. 20318-8000

Commanding Officer 1
Naval Research Laboratory
Code 5150
Washington, D.C. 20375-5000

Defense Advanced Research Projects Agency (DARPA) 1
Integrated Strategic Technology Office (ISTO)
1400 Wilson Boulevard
Arlington, VA 22209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Naval Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Prototype Projects Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

Defense Advanced Research Projects Agency (DARPA) 1
Director, Tactical Technology Office
1400 Wilson Boulevard
Arlington, VA 2209-2308

Dr. R. M. Carroll (OP-01B2) 1
Chief of Naval Operations
Washington, DC 20350

Dr. Aimram Yehudai 1
Tel Aviv University
School of Mathematical Sciences
Department of Computer Science
Tel Aviv, Israel 69978

Dr. Bernd Kraemer 1
GMD Postfach 1240
Schloss Birlinghaven
D-5205
Sankt Augustin 1, West Germany

Dr. Robert M. Balzer 1
USC-Information Sciences Institute
4676 Admiralty Way
Suite 1001
Marina del Ray, CA 90292-6695

Dr. Ted Lewis 1
Editor-in-Chief, IEEE Software
Oregon State University
Computer Science Department
Corvallis, OR 97331

IBM T.J.Watson Research Center 1
Attn. Dr. A. Stoyenko
P.O. Box 704
Yorktown Heights, NY 10598

Dr. R. T. Yeh
International Software Systems Inc. 1
12710 Research Boulevard, Suite 301
Austin, TX 78759

Attn. Dr. C. Green 1
Kestrel Institute
1801 Page Mill Road
Palo Alto, CA 94304

Prof. D. Berry 1
Department of Computer Science
University of California
Los Angelas, CA 90024

Dr. B. Liskov 1
Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
545 Tech Square
Cambridge, MA 02139

Dr. J. Guttag 1
Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
545 Tech Square
Cambridge, MA 02139

Director, Naval Telecommunications System Integration Center 1
NAVCOMMUNIT Washington
Washington, D.C. 20363-5110

Space and Naval Warfare Systems Command 1
Attn: Dr. Knudsen, Code PD50
Washington, D.C. 20363-5110

Ada Joint Program Office 1
OUSDRE(R&AT)
The Pentagon
Washington, D.C. 23030

CAPT A. Thompson 1
Naval Sea Systems Command
National Center #2, Suite 7N06
Washington, D.C. 22202

Dr. Peter Ng 1
New Jersey Institute of Technology
Computer Science Department
Newark, NJ 07102

Dr. Van Tilborg 1
Office of Naval Research
Computer Science Division, Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. R. Wachter 1
Office of Naval Research
Computer Science Division, Code 1133
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. J. Smith, Code 1211 1
Office of Naval Research1
Applied Mathematics and Computer Science
800 N. Quincy Street
Arlington, VA 22217-5000

Dr. R. Kieburtz 1
Oregon Graduate Center1
Portland (Beaverton)
Portland, OR 97005

Dr. M. Ketabchi 1
Santa Clara University
Department of Electrical Engineering and Computer Science
Santa Clara, CA 95053

Attn. Dr. L. Belady 1
Software Group, MCC
9430 Research Boulevard
Austin, TX 78759

Attn. Dr. Murat Tanik 1
Southern Methodist University
Computer Science and Engineering Department
Dallas, TX 75275

Dr. Ming Liu 1
The Ohio State University
Department of Computer and Information Science
2036 Neil Ave Mall
Columbus, OH 43210-1277

Mr. William E. Rzepka 1
U.S. Air Force Systems Command
Rome Air Development Center
RADC/COE
Griffis Air Force Base, NY 13441-5700

Dr. C.V. Ramamoorthy 1
University of California at Berkeley
Department of Electrical Engineering and Computer Science
Computer Science Division
Berkeley, CA 90024

Dr. Nancy Levenson 1
University of California at Irvine
Department of Computer and Information Science
Irvine, CA 92717

Dr. Mike Reiley 1
Fleet Combat Directional Systems Support Activity
San Diego, CA 92147-5081

Dr. William Howden 1
University of California at San Diego
Department of Computer Science
La Jolla, CA 92093

Dr. Earl Chavis (OP-162) 1
Chief of Naval Operations
Washington, DC 20350

Dr. Jane W. S. Liu 1
University of Illinois
Department of Computer Science
Urbana Champaign, IL 61801

Dr. Alan Hevner
University of Maryland
College of Business Management
Tydings Hall, Room 0137
College Park, MD 20742 — 1

Dr. Y. H. Chu
University of Maryland1
Computer Science Department
College Park, MD 20742 — 1

Dr. N. Roussapoulos
University of Maryland
Computer Science Department
College Park, MD 20742 — 1

Dr. Alfs Berztiss
University of Pittsburgh
Department of Computer Science
Pittsburgh, PA 15260 — 1

Dr. Al Mok
University of Texas at Austin
Computer Science Department
Austin, TX 78712 — 1

George Sumiall
US Army Headquarters
CECOM
AMSEL-RD-SE-AST-SE
Fort Monmouth, NJ 07703-5000 — 1

Attn: Joel Trimble
1211 South Fern Street, C107
Arlington, VA 22202 — 1

Naval Ocean Systems Center
Attn: Linwood Sutton, Code 423
San Diego, CA 92152-5000 — 1

LuQi 30
Code 52Lq
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Valdis Berzins 1
Code 52Be
Naval Postgraduate School
Monterey, CA 93943

Major James M. Huskins 1
PEO STAMIS
Ft. Belvoir, VA 22060

Patrick D. Barnes 10
Code 52 Ba
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943