AD-A255 656

ADA-BASED MULTITASKING TERMINAL I/O

BY JOHN C. CHALKLEY   MICHAEL W. MASTERS
COMBAT SYSTEMS DEPARTMENT

FEBRUARY 1992

DTIC
ELECTE
SEP. 14 1992
S   D

NAVAL SURFACE WARFARE CENTER

Dahlgren, Virginia 22448-5000 • Silver Spring, Maryland 20903-5000

92-25054

92 9 10 102

ADA-BASED MULTITASKING TERMINAL I/O

BY JOHN C. CHALKLEY   MICHAEL W. MASTERS
COMBAT SYSTEMS DEPARTMENT

FEBRUARY 1992

NAVAL SURFACE WARFARE CENTER
Dahlgren, Virginia 22448-5000 ● Silver Spring, Maryland 20903-5000

FOREWORD

This report describes a set of reusable Ada packages designed by NSWCDD to perform flexible terminal I/O functions in an Ada multitasking application programming environment. Many terminal I/O services offered by Ada compiler vendors (e.g., the TEXT_IO package required by the Ada Language Reference Manual) do not provide adequate concurrency primatives to support use in multitasking programs. In particular, they are process synchronous rather than task synchronous/process asynchronous, causing all program activity to block during waits for keyboard input. Thus, they cannot be used effectively in real-time systems.
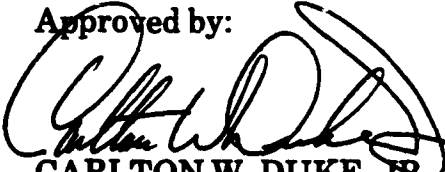
This software, called "ANSI Terminal Services," provides nonblocking terminal I/O and full concurrency protection for multitasking operations, including use of multiple terminals from a single application program. ANSI Terminal Services is designed to be used with a wide range of compilers, operating systems, and terminal hardware configurations, with a minimum of modification. Currently supported operating systems are DEC VAX VMS and SUN OS 4.1 (BSD 4.3 UNIX). Development of an MS-DOS version would be straightforward. Terminal hardware customization is supplied in the form of user-defined files that permit use with a wide variety of terminals. There is virtually no compiler-unique dependency in the design.

The development of ANSI Terminal Services was initially funded by a NAVSWC AEGIS Engineering Initiatives Study, which investigated using Ada for future AEGIS baselines. Since its initial development, it continues to be used in a variety of applications, including in the NAVSWC AEGIS CDS Distributed Architecture Experiment (CDAE), in the joint DARPA/AEGIS High-Performance Distributed Computing Project (HiPer-D), and in an analyst's testbed that is being used for development of new TOMAHAWK Track Control Group (TCG) mathematical algorithms. This report is intended to provide full documentation in order to foster further use of the software.

The authors acknowledge the considerable contribution of Catherine Ray, who developed early versions of several components of ANSI Terminal Services, including a line editor and certain keyboard input features. The authors also acknowledge the contribution of Harry Leung, who developed the low-level interface to SUN OS 4.1.

This report has been reviewed by Richard Stutler, Head, Combat System Technologies Branch, and R. Neal Cain, Head, Engineering and Technology Division.

Approved by:

CARLTON W. DUKE, JR., Head
Combat Systems Department

ABSTRACT

This report describes "ANSI Terminal Services," a reusable Ada-based layered approach to providing protected asynchronous terminal I/O in a multitasking environment. The traditional single-threaded, sequential programming model is inadequate for Ada multitask programs, particularly for real-time applications. An application is likely to contain several tasks needing to perform output concurrently. Any shared resources contained in the I/O software must, therefore, be protected through some form of mutual exclusion. Also, an application will likely need to support input operations concurrently with outputs. This requirement is especially important since Ada implementations on most current operating systems are process synchronous, rather than task synchronous/process asynchronous. As a result, they cause the entire user process to block on an input request. Thus, other program tasks that may be performing critical functions are suspended until user input is complete. Such a result is clearly inappropriate in a real-time environment. ANSI Terminal Services solves these problems and provides both a standard programmer interface and the necessary multitasking concurrency control.

DTIC QUALITY INSPECTED 3

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

CONTENTS

CONTENTS (Continued)

ILLUSTRATIONS

TABLES

CHAPTER 1

INTRODUCTION

Terminal I/O in a multithreaded programming environment, such as that supported by the task feature of the Ada programming language, engenders special considerations. The traditional single-threaded, sequential programming model (i.e., program-wide execution suspension or blocking to wait for keyboard input and screen scrolling of output) is inadequate for Ada multitask programs, particularly for real-time applications.

Typically, multitasking programs differ in a variety of ways. First, within the application program there are likely to be several, perhaps many tasks that need to do output concurrently. Any shared resources contained in the I/O software must, therefore, be protected through some form of mutual exclusion. At a minimum, cursor state is a shared resource in such situations.

Second, the application program will likely need to support input operations concurrently with outputs. This requirement is especially important since Ada implementations on most current operating systems (e.g., VMS, UNIX, and MS-DOS) are process synchronous rather that task synchronous/process asynchronous. As a result, they cause the entire user process to block while the application waits for input from the program user. Thus, other program tasks that may be performing absolutely critical functions are suspended until user input is complete.

Third, real-time multitasking terminal use normally involves display screens that map different program outputs to fixed locations or regions (panels) of the screen. Thus, the values of critical status parameters may be displayed and updated upon change in one or more screen panels: time of day, elapsed time, and other critical parameters may be updated periodically in other panels; the user may be prompted for text input in another panel; and a menu providing a number of program control options, selectable via function or control keys, may be displayed in another panel.

Figure 1-1 illustrates this situation. The user should be able to perform normal editing functions concurrently with program-driven output to other panels. However, upon input of a menu function key, the terminal handling software should also be able to abandon user input at the text prompt and immediately initiate the
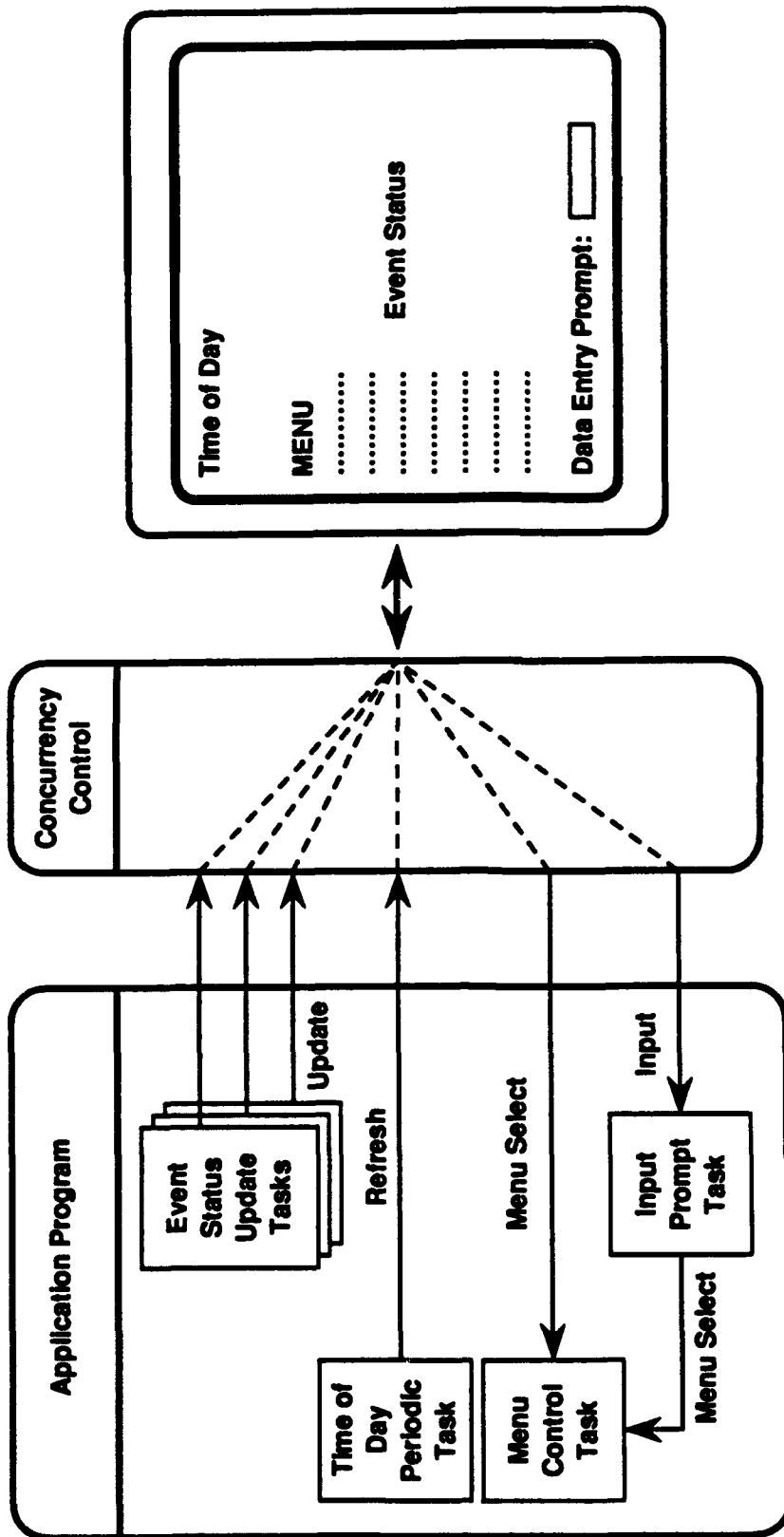
FIGURE 1-1. MULTITASK DISPLAY PROGRAM

requested menu selection. In such applications, the screen as a whole must not be allowed to scroll, although scrolling within some panels may be desired.

1.1 DESIGN CONCEPT

Some developers have provided useful reusable software packages that support flexible terminal I/O, while ignoring the mechanisms needed to insure nonblocking input and proper protection of shared resources in a multitasking programming environment. In effect, these solutions leave the most difficult part of the problem to the application developer, that of insuring that their products work properly in the presence of multiple user tasks. The software described in this report approaches the problem differently. It begins with the assumption that it is to be used in a multitasking environment and, from the beginning, builds in the necessary mechanisms for concurrency.

This approach has resulted in a layered set of terminal I/O Ada packages, where input and output operations are built up step-by-step from low-level, system-dependent I/O services, through successive abstractions, to a high-level user interface that provides designed-in concurrency. This approach is shown in Table 1-1.

TABLE 1-1. LAYERED APPROACH TO MULTITASKING TERMINAL I/O

Ada Multitasking Application
ANSI Terminal Services Programmer Interface
ANSI Terminal Services Concurrency Control

Intermediate I/O Services
    • Location-specified screen output
    • Cursor control via ANSI escape sequences
    • Line editing of text entry
    • Basic abstracted keyboard input
    • Bindings to low-level I/O

Operating System Specific Nonblocking I/O

Note that the highest layer, called "ANSI Terminal Services" throughout this report, provides both a standard programmer interface and the necessary multitasking concurrency control. Lower, nonconcurrent levels are effectively hidden, but can be used directly, if needed, by a knowledgeable designer. The bottom layer, nonblocking I/O services, is operating system specific. One of the key features

of the layered design is localization and minimization of these dependencies in order to support portability and reuse to the maximum extent possible.

1.2   TOP-LEVEL REQUIREMENTS

The multitasking operating environment described above, combined with goals of portability and reusability, lead to a number of top-level requirements for the terminal I/O capability illustrated in Table 1-1. These requirements primarily address functional capabilities, but the need for portable and reusable software has an additional impact on design. Each requirement is first listed below, then further elaborated in the paragraphs that follow. Chapter 2 and Appendix A provide a complete and definitive specification of program interface and functional capabilities, including:

- Portable, reusable Ada design with minimum dependencies on terminal hardware, operating system services, and compiler-specific features

- Built-in mutual exclusion and nonblocking I/O support, including timed input operations, for Ada multitasking applications

- Support for *fixed screen* input and output using standard ANSI *escape sequences*, including color when available

- Keyboard independence, including a mechanism to dynamically bind edit keys, function keys, etc., at runtime

- Standard keyboard input features, including line editing, timed input, and immediate input termination on detection of function and control keys

- Use of multiple terminals from the same program

- Use of terminal ports for direct intercomputer I/O

1.2.1   Portable Ada Design

The goal of portability and reusability is, strictly speaking, not one of functionality, but rather of design. This goal is achieved primarily through the proper choice of layering. The layered approach was introduced in the previous section. Table 1-2 lists hardware and operating system dependencies, together with the recommended mechanism for porting to alternate implementations.

TABLE 1-2. DEPENDENCIES AND RECOMMENDED PORTING MECHANISM

DEPENDENCY	LOCATION	PORTING REQUIREMENT
Keyboard escape sequence values	Keyboard input package	User-definable dynamic key mappings read from user-specified file.
Operating system I/O services	Asynchronous I/O package	Rewrite of package body to use OS-specific nonblocking I/O services.
Cursor control escape sequences	Cursor services package	Rewrite of escape sequence data declarations in package body.
Limit values	Constants package	Limit values input from user-specified file during elaboration.
Task priorities	ANSI Terminal Services package	Package recompilation (Ada task priorities must be static).

1.2.2 Multitasking, Nonblocking I/O

In addition to providing a functional capability, support for multitasking during I/O operations impacts design as well. In particular, it implies the presence of some form of mutual exclusion to protect shared resources during input and output operations. Specifically, the location and color attributes of the cursor must be protected during I/O by multiple application tasks. Without this protection, it is impossible to insure the integrity of screen output and of keyboard input. As will be shown in Chapter 3, this protection is provided by two Ada tasks: a screen control task and a keyboard control task. These two tasks interact during input operation, because the screen task is used to echo printable *keystrokes* upon input via the keyboard.

Nonblocking I/O support is operating system dependent. This may be accomplished in three ways. The first is through some form of interrupt mechanism, such as software traps or signals. This method might more accurately be called asynchronous I/O. Some operating system vendors provide Ada bindings to such services, in which case this approach is to be preferred. For example, Digital Equipment Corporation (DEC) provides Ada packages (STARLET, TASKING_ SERVICES, CONDITION_HANDLING), which permit use of VMS operating system asynchronous system traps (AST).

The second method involves polling the input channel to determine if any *keystrokes* have been entered. Any available input is processed. If no input is

available, a delay is executed, followed by another *polling* operation. This method has two disadvantages. First, the *polling* operation consumes CPU time if there is no data available; and second, a *keystroke* arriving just after execution of the delay operation will not be detected and processed until after the delay expires, thus introducing a small latency. Choice of a suitable polling interval is critical to performance.

Ada implementers on UNIX systems have been less inclined to supply organic asynchronous input support, leaving application developers the problem of building their own SIGNAL-handling routines. The current release of ANSI Terminal Services for SUN OS is a polling implementation.

The third method for achieving nonblocking I/O is through use of an operating system that supports multiple threads of execution within the user's process. In such operating systems, waiting for input in one thread will not block program execution in any other thread. The MACH operating system and its derivative, OSF-1, are examples. The POSIX operating system interface standard specifies a similar capability, called "p-threads," although POSIX/p-thread-compliant operating systems are only now beginning to appear in the commercial arena.

1.2.3   ANSI Standard Fixed Screen Color Output

There is an ANSI standard for *escape sequences* used to control terminal cursor location and attributes; e.g., line-wrap mode (*on* or *off*), blink setting, reverse video, bold text, and color. Virtually all terminals use this standard, although not all terminals support color. ANSI Terminal Services makes use of these ANSI *escape sequences*. Nonstandard terminals are not supported. In particular, to achieve *fixed screen* output stability, line-wrap mode is automatically set to *off* to prevent wrap or scrolling.

1.2.4   Keyboard Independence

Although ANSI *escape sequences* provide a widely accepted standard for screen output, no such standardization exists for keyboards. Virtually every computer and terminal vendor provides unique products, either in keys present on the keyboard or in *escape sequences* generated by individual *keystrokes* or both. The printable and control keys, of course, generate standard ASCII key codes in the range of zero to 127. But edit keys and function keys follow no such standard, either as to key name or *escape sequence* generated.

Providing a mechanism to permit dynamic binding of *keystroke* to *escape sequences* is crucial to the portability of software across terminal hardware. The terminal I/O services described in this report support dynamic binding by means of a

user-specified file containing the mapping of edit and function keys to *escape sequences*.

1.2.5 Standard Keyboard Input Operations

A number of standard input operations are desirable. These include the ability to perform line editing using standard edit keys; the ability to read all *keystrokes* supported by the keyboard, including function keys; and the ability to perform timed input. In the last case, the input operation should terminate either on completion of the requested input or expiration of the specified time interval, with suitable indication of which termination mode was experienced. Finally, in the case of both timed- and line-edit operations, immediate termination of input must be performed on the detection of function and control keys.

1.2.6 Use of Multiple Terminals

In many applications, it is useful to be able to control many terminals from the same program. Most operating systems provide services to assign multiple terminal ports to the same user process. In some cases, this operation must be accompanied by system privileges beyond those granted to a normal user process. Further, it is also useful that access to each port be bound at runtime to programmer-referenceable data structures so that terminal identity can be tested for, passed as a subprogram calling parameter, and assigned as a component of other data structures (e.g., as a component of an array). ANSI Terminal Services provides this mechanism by means of an Ada private type that is allocated at runtime when the operating system port assignment service is called.

1.2.7 Direct Intercomputer I/O

Terminal ports may be used to perform intercomputer I/O in addition to normal screen output and keyboard input operations. In this case, the use of cursor control on output, and *escape sequence* parsing on input are bypassed. Otherwise, the mutual exclusion mechanisms supplied to support multitasking programming are still present. Note that since no cursor control is applied, if this mode of I/O is used in conjunction with a terminal, and if the output string includes a carriage return, a traditional scrolling screen display can be accomplished.

1.3 REPORT ORGANIZATION

The remainder of this report is divided into five additional chapters. Chapter 2 provides a detailed specification of ANSI Terminal Services capabilities. It is the programmer's interface guide. It deals only with the package specification of the

highest layer of software, package ANSI_TERMINAL_SERVICES. Lower layers are discussed in Chapter 3. In addition to discussing the specifications of lower-level packages, this third chapter also describes the details of the implementations contained in each package body, including ANSI_TERMINAL_SERVICES.

Chapter 4 illustrates the use of ANSI Terminal Services in a concise example. Chapter 5 provides a limited discussion of the execution performance characteristics of this software. It should be cautioned that performance is highly system dependent. Users should rely on their own benchmark tests, where performance is an issue. Finally, Chapter 6 briefly discusses areas that may experience further development in the future.

CHAPTER 2

FUNCTIONAL DEFINITION

This section specifies and defines the capabilities of a set of asynchronous I/O packages written in Ada. Collectively, these packages provide a highly reusable means for protected terminal I/O in a multitasking environment. For instance, users may safely perform I/O such as line editing and screen updating within concurrent Ada tasks. Specifically, these packages include the following capabilities:

- Convenient routines for assigning ports
- A flexible means of mapping any keyboard at runtime
- I/O routines that protect critical resources shared among Ada tasks
- Means of imposing time constraints upon read requests
- Color support for adequately equipped terminals
- Control over an arbitrary number of terminals

ANSI_TERMINAL_SERVICES is the single package that provides users with the above capabilities. This chapter documents the specification of this package and is organized as follows:

- Overview of package hierarchy and capability
- Visible data types
- Private data types
- Terminal allocation and deallocation
- Output services
- Input services
- Convenience functions
- Elaborated constants
- Exception handling

2.1 OVERVIEW

The function and the purpose of ANSI_TERMINAL_SERVICES and its five underlying support packages are briefly outlined in Sections 2.1.1 through 2.1.6. The application interface and the interrelationships among this set of packages are illustrated in Figure 2-1. While the functionality of ANSI_TERMINAL_SERVICES is fully described here in Chapter 2, the supporting packages are documented in Chapter 3, as referenced in Table 2-1.
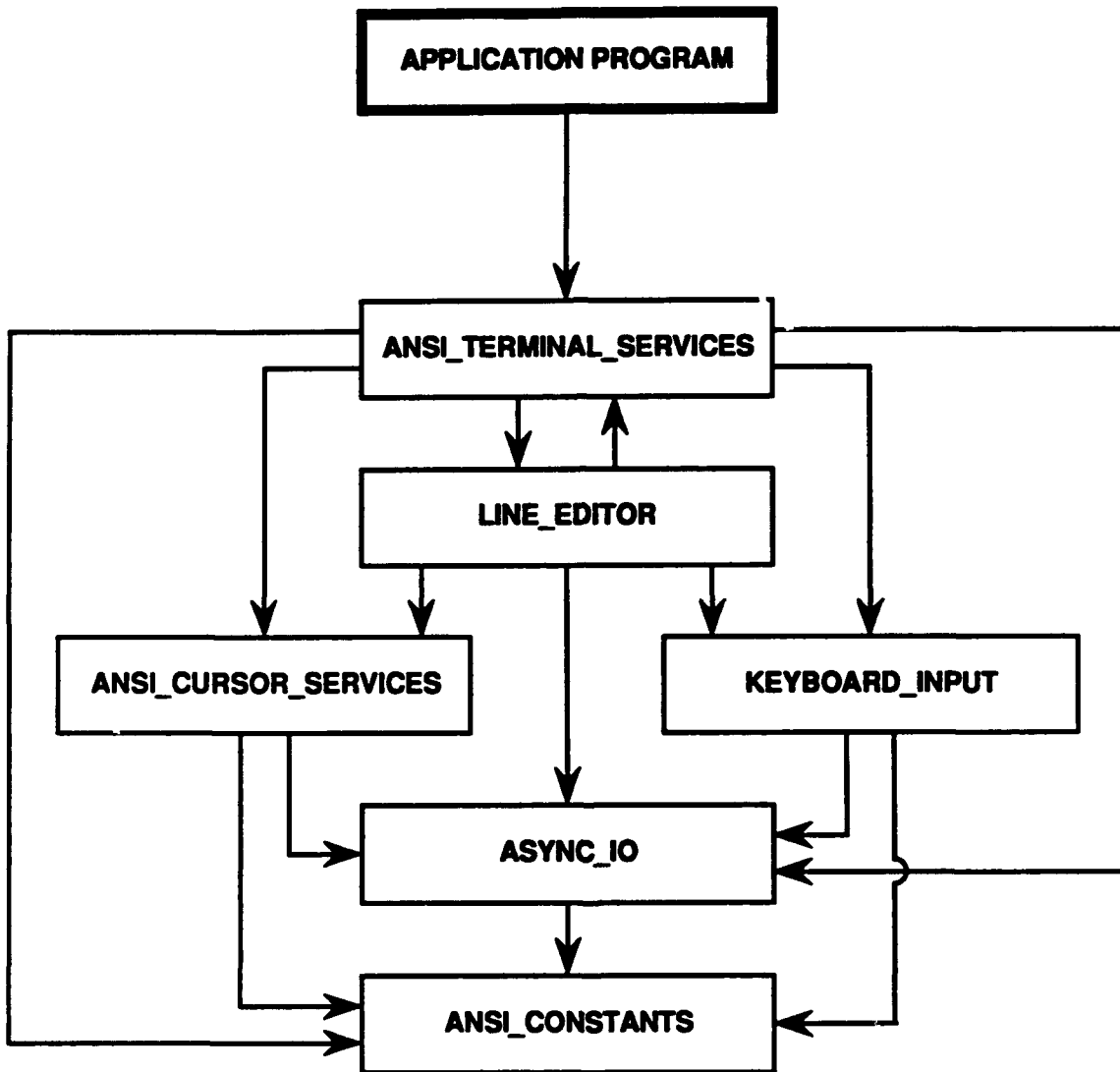
FIGURE 2-1. PACKAGE DEPENDENCY DIAGRAM

TABLE 2-1. SUPPORTING PACKAGES

PACKAGE NAME	REFERENCE
ANSI_CURSOR_SERVICES	3.2
LINE_EDITOR	3.3
KEYBOARD_INPUT	3.4
ASYNC_IO	3.5
ANSI_CONSTANTS	3.6

2.1.1   ANSI Terminal Services

ANSI_TERMINAL_SERVICES addresses the problems inherent to concurrent I/O and provides convenient, safe mechanisms for performing terminal I/O among multiple Ada tasks. Specifically, ANSI_TERMINAL_SERVICES defines a terminal resource manager that provides the synchronization necessary to perform I/O in a multitasking environment on 7-bit terminals. This top-level package also provides the complete interface users will need to access the functionality of all supporting packages. As described in Section 2.2, ANSI_TERMINAL_SERVICES derives all data types and interfaces with procedures declared in lower packages so that applications are dependent only upon this one package. Although the dependencies have been removed, all functionality of the lower packages has been retained.

2.1.2   ANSI Cursor Services

ANSI_CURSOR_SERVICES includes the constants and types needed to describe the cursor or a display. This supporting package also includes procedures that use ANSI *escape sequences* to modify cursor and screen appearance. ANSI_TERMINAL_SERVICES provides the necessary interface to this package so that applications will not be dependent upon it.

2.1.3   Line Editor

LINE_EDITOR contains procedures to implement a simple yet flexible line editor that is not dependent upon any particular keyboard. It has the capability of presenting a default string that may be either accepted or edited by the user.

Applications may access this editor through the top-level package ANSI_
TERMINAL_SERVICES.

2.1.4   Keyboard Input

KEYBOARD_INPUT defines an abstraction of a keyboard, which may then be
mapped to any physical keyboard. This binding is done at runtime via a keyboard
mapping file (See Appendices B and C). Dependent packages (LINE_EDITOR and
ANSI_TERMINAL_SERVICES) operate upon KEYBOARD_INPUT's abstracted
keys without concern for the underlying ASCII representation. In this manner,
KEYBOARD_INPUT makes this entire set of Ada packages reusable. Because of
this design, the act of porting an application to another terminal should only involve
changing the keyboard bindings, rather than recompiling source code. As before,
applications may access KEYBOARD_INPUT's functionality indirectly through
ANSI_TERMINAL_SERVICES.

2.1.5   Async IO

ASYNC_IO is the lowest-level I/O package in this collection. Although the
package specification is applicable to almost any set of underlying operating system
terminal I/O services, the body of ASYNC_IO contains I/O routines which are largely
system dependent. One version of this package supports DEC's VAX/VMS system
calls, for example, while another supports UNIX. In either case, the functionality is
equivalent; ASYNC_IO initializes I/O channels and performs I/O services needed by
other packages in this collection. As with ANSI_CURSOR_SERVICES, LINE_
EDITOR, and KEYBOARD_INPUT, this package should not be visible to an
application because its functionality is encapsulated within ANSI_TERMINAL_
SERVICES.

2.1.6   ANSI Constants

ANSI_CONSTANTS initializes critical constants at runtime. During package
elaboration, this package opens the ANSI.DAT file and reads several integer values.
The higher packages, in turn, initialize their constants with the values from the file.
This allows users to customize ANSI_TERMINAL_SERVICES to their needs without
the need to recompile any source code. The constants initialized in this manner are
tabulated in Section 2.8. Appendix D describes the simple format of the constants
file, and Appendix E shows the contents of the default file.

2.2 VISIBLE DATA TYPES

When performing terminal I/O in a multitasking environment, an application needs data types providing the following capabilities:

- Port assignment and identification
- Keyboard *layout* specification
- Cursor placement
- Cursor attribute assignment
- Formatted text specification

ANSI_TERMINAL_SERVICES provides the data types described in Sections 2.2.1 through 2.2.16 to meet the above needs. All of these types actually originate in packages other than ANSI_TERMINAL_SERVICES. In an effort to decrease an application's reliance upon multiple packages, all important data types from the supporting packages are derived or redeclared in ANSI_TERMINAL_SERVICES. With this simplified interface, a user needs only to make ANSI_TERMINAL_SERVICES visible to the application without including any supporting packages. Instead of *WITH*ing the KEYBOARD_INPUT package to gain access to a certain type, for instance, a user references the corresponding derived type in ANSI_TERMINAL_SERVICES.

Unfortunately, Ada does not provide a convenient means of deriving record types so that their individual components are also derived. If a record type from ANSI_CURSOR_SERVICES is derived in ANSI_TERMINAL_SERVICES, for example, the components of the new type will be of a type from the lower package. In order to provide an interface with these records that is consistent with that of the derived types described above, ANSI_TERMINAL_SERVICES redeclares important record types from its supporting packages. These new types are identical in structure and name to the original records. All explicit type conversions needed to implement these new types are localized within ANSI_TERMINAL_SERVICES and should be of no concern to the user. Table 2-2 lists both the derived types and the redeclared record types, along with their originating packages.

2.2.1 Port Name

To identify a terminal port by name in ANSI_TERMINAL_SERVICES, users should use the PORT_NAME type. This type is simply a string containing a port name that is valid for the present terminal (e.g., TXA1:, ttya). The string is constrained in length by the constant MAX_PORT_NAME_SIZE. If the default length (15) is inappropriate, users may modify its value in the constants file (see Section 2.8).

TABLE 2-2. TYPES DERIVED OR REDECLARED IN ANSI_TERMINAL_SERVICES

| TYPE | ORIGINATING PACKAGE | ANSI_TERMINAL_SERVICES | | REFERENCE |
		DERIVED	REDECLARED	
PORT_NAME	ASYNC_IO	✓		2.2.1
PORT_DATA	ASYNC_IO	✓		2.2.2
ALL_KEYS	KEYBOARD_INPUT	✓		2.2.3
ASCII_KEYS	KEYBOARD_INPUT	✓		2.2.3
CTRL_KEYS	KEYBOARD_INPUT	✓		2.2.3
PRINTABLE_KEYS	KEYBOARD_INPUT	✓		2.2.3
UC_LETTERS	KEYBOARD_INPUT	✓		2.2.3
LC_LETTERS	KEYBOARD_INPUT	✓		2.2.3
NUMBER_KEYS	KEYBOARD_INPUT	✓		2.2.3
MAPPED_KEYS	KEYBOARD_INPUT	✓		2.2.3
EDIT_KEYS	KEYBOARD_INPUT	✓		2.2.3
FUNCTION_KEYS	KEYBOARD_INPUT	✓		2.2.3
MAP_LIST	KEYBOARD_INPUT	✓		2.2.4
LINE	ANSI_CURSOR_SERVICES	✓		2.2.5
COLUMN	ANSI_CURSOR_SERVICES	✓		2.2.6
COLUMN_INCREMENT	ANSI_CURSOR_SERVICES	✓		2.2.8
SCREEN_COLORS	ANSI_CURSOR_SERVICES	✓		2.2.9
INTENSITY_SETTING	ANSI_CURSOR_SERVICES	✓		2.2.10
BLINK_SETTING	ANSI_CURSOR_SERVICES	✓		2.2.11
V_STRING	ANSI_CURSOR_SERVICES	✓		2.2.14
LAYOUT_INDEX	ANSI_CURSOR_SERVICES	✓		2.2.16
POSITION	ANSI_CURSOR_SERVICES		✓	2.2.7
TEXT_COLOR	ANSI_CURSOR_SERVICES		✓	2.2.12
ATTRIBUTE	ANSI_CURSOR_SERVICES		✓	2.2.13
TEXT_REC	ANSI_CURSOR_SERVICES		✓	2.2.15
LAYOUT	ANSI_CURSOR_SERVICES		✓	2.2.16

2.2.2   Port Description

To aid in the assignment of a terminal port, ANSI_TERMINAL_SERVICES provides the PORT_DATA record type (see Table 2-3). This type is an encapsulation of the requested port name and an assigned index into an internal channel array of a system-dependent data type. When initializing a terminal, applications simply supply a value of type PORT_NAME (2.2.1). The supporting package ASYNC_IO subsequently provides access to this port and assigns a value to PORT_DATA's channel index component. This index is of the private subtype CHANNEL_TYPE (3.5.1) and includes integers in the range 1 to 16. If more than 16 channel indexes are required for an application, the constant MAX_CHANNELS may be customized by altering the constants file (see Section 2.8). Normally, of course, applications should have no concern with the assigned channel or its index. These constructs are internal to ANSI_TERMINAL_SERVICES and ASYNC_IO and are set when the user names a requested port.

TABLE 2-3. PORT IDENTIFICATION RECORD (TYPE PORT_DATA)

COMPONENT	TYPE	LEGAL VALUES	ASSIGNMENT	REFERENCE
Port_ID	PORT_NAME	string (1..MAX_PORT_NAME_SIZE)	By application	2.2.1
Channel	CHANNEL_TYPE	1..MAX_CHANNELS	By ASYNC_IO	3.5.1

2.2.3   Logical Keys

Since ANSI_TERMINAL_SERVICES was designed to be highly reusable, it considers the fact that all keyboards are not identical. Some have ten function keys, while others have twenty, or none at all. Similarly, some keyboards allow separate meta-key combinations such as **Shift-F1** or **Control-F1**. Another common disparity involves keys related to editing functions. Some keyboards have keys labeled **Insert** and **Erase Line**, while others do not. ANSI_TERMINAL_SERVICES addresses this problem with a collection of *logical keys*.

A *keystroke* or *logical key* should be considered an abstraction of the keyboard and can represent one or more ASCII characters. For example, if a user presses a key corresponding to a printable ASCII character, then the *keystroke* is simply a representation of that individual character. However, the user may press a function or edit key that generates a series of ASCII characters prefixed by an escape character (ASCII 27). This entire sequence of characters (usually three to five

characters in length) is commonly called an *escape sequence* and also corresponds to a single *logical key*.

Type ALL_KEYS is originally declared in KEYBOARD_INPUT and is derived in ANSI_TERMINAL_SERVICES. It is merely an enumerated type listing 242 possible *logical keys*. The contents of ALL_KEYS are listed below in their proper enumerated order:

(ASCII Control Characters:)

NUL,				
CTRL_A,	CTRL_B,	CTRL_C,	CTRL_D,	CTRL_E,
CTRL_F,	CTRL_G,	CTRL_H,	CTRL_I,	CTRL_J,
CTRL_K,	CTRL_L,	CTRL_M,	CTRL_N,	CTRL_O,
CTRL_P,	CTRL_Q,	CTRL_R,	CTRL_S,	CTRL_T,
CTRL_U,	CTRL_V,	CTRL_W,	CTRL_X,	CTRL_Y,
CTRL_Z,	ESC,	FS,	GS,	RS,
US,				

(ASCII Printable Characters:)

SPC,	EXCLAM,	QUOTATION,	SHARP,
DOLLAR,	PERCENT,	AMPERSAND,	SINGLE_QUOTE,
L_PAREN,	R_PAREN,	ASTERISK,	PLUS,
COMMA,	MINUS,	PERIOD,	SLASH,

ZERO,	ONE,	TWO,	THREE,	FOUR,
FIVE,	SIX,	SEVEN,	EIGHT,	NINE,

COLON,	SEMICOLON,	LESS_THAN,	EQUAL,
GREATER_THAN,	QUERY,	AT_SIGN,	

UC_A,	UC_B,	UC_C,	UC_D,	UC_E,	UC_F	UC_G,
UC_H,	UC_I,	UC_J,	UC_K,	UC_L,	UC_M,	UC_N,
UC_O,	UC_P,	UC_Q,	UC_R,	UC_S,	UC_T,	UC_U,
UC_V,	UC_W,	UC_X,	UC_Y,	UC_Z,		

L_BRACKET,	BACK_SLASH,	R_BRACKET,
CIRCUMFLEX,	UNDERLINE,	GRAVE,

LC_A,	LC_B,	LC_C,	LC_D,	LC_E,	LC_F	LC_G,
LC_H,	LC_I,	LC_J,	LC_K,	LC_L,	LC_M,	LC_N,
LC_O,	LC_P,	LC_Q,	LC_R,	LC_S,	LC_T,	LC_U,
LC_V,	LC_W,	LC_X,	LC_Y,	LC_Z,		

L_BRACE,	BAR,	R_BRACE,	TILDE,	DEL,

(Edit keys:)

DELETE,	INSERT,	HOME,	ENDD,
ERASE_LINE,	BACKSPACE,	LEFT_ARROW,	RIGHT_ARROW,
TAB,	TAB_REVERSE,	PAGE_DOWN,	PAGE_UP,
UP_ARROW,	DOWN_ARROW,		

(Function keys:)

FK1, FK2, FK3, FK4, FK5, FK6, FK7, FK8, FK9, FK10,
FK11, FK12, FK13, FK14, FK15, FK16, FK17, FK18, FK19, FK20,
FK21, FK22, FK23, FK24, FK25, FK26, FK27, FK28, FK29, FK30,
FK31, FK32, FK33, FK34, FK35, FK36, FK37, FK38, FK39, FK40,
FK41, FK42, FK43, FK44, FK45, FK46, FK47, FK48, FK49, FK50,
FK51, FK52, FK53, FK54, FK55, FK56, FK57, FK58, FK59, FK60,
FK61, FK62, FK63, FK64, FK65, FK66, FK67, FK68, FK69, FK70,
FK71, FK72, FK73, FK74, FK75, FK76, FK77, FK78, FK79, FK80,
FK81, FK82, FK83, FK84, FK85, FK86, FK87, FK88, FK89, FK90,
FK91, FK92, FK93, FK94, FK95, FK96, FK97, FK98, FK99, FK100

Most of these *logical keys* correspond directly to the ASCII character set supported by all keyboards. For example, ALL_KEYS includes entries such as AMPERSAND, L_BRACKET, LC_A, and UC_A to match the ASCII characters &, [, a, and A, respectively. It should be noted from the list above that the ordering of the elements within type ALL_KEYS is not arbitrary. Those *logical keys*, corresponding to the 128 ASCII characters, are positioned such that they parallel the ordering exhibited by the ASCII character set. For example, *logical key* UC_A occupies position 65 in type ALL_KEYS, and the ASCII character A corresponds to position 65 in the character set.

In addition to the 128 ASCII *logical keys*, ALL_KEYS includes 14 edit *keystrokes* (DELETE .. DOWN_ARROW) common to most keyboards. Since such edit keys are not standardized, the physical keys may generate different *escape sequences* on different keyboards. Their corresponding *logical keys*, however, may be mapped to whatever *escape sequence* the actual keys generate (see Section 2.2.4). Of course, there may not even be a corresponding physical key. In this case, the *keystroke* could be mapped to any physical key, such as a function key, that generates an *escape sequence*. If a keyboard has no **Home** key, for example, the *logical key* HOME could easily be mapped to the key labeled **F1**. Alternatively, the edit *logical key* could be mapped to a control character if no corresponding physical key exists. For example, HOME could be mapped to **Control-H** if a keyboard has no key labeled **Home**. Users will notice that the three *logical keys* TAB, BACKSPACE, and DELETE normally are equivalent to ASCII 9, 8, and 127, respectively. By default, these *keystrokes* will be mapped in this manner. If needed, these mappings can easily be changed to fit any user's preferences or any keyboard *layout* (see Section 2.2.4).

For greater flexibility, ALL_KEYS also includes 100 extra *logical function keys*, which can be bound to any *escape sequence*. These extra *keystrokes* (FK1, FK2, ..., FK100) are commonly associated with function keys, meta-combinations (e.g., **Shift-F2**), and edit keys that have no direct equivalent within ALL_KEYS. Given these 100 extra *logical keys*, type ALL_KEYS should cover virtually all possible *keypresses* a keyboard supports.

KEYBOARD_INPUT partitions type ALL_KEYS by declaring the nine subtypes of Table 2-4. These subtypes group the *logical keys* functionally. For example, type UC_LETTERS consists of the *logical keys* corresponding to the uppercase letters. As shown in Figure 2-2, these new types also exhibit a natural hierarchy. The *logical key* SIX, for instance, is a member of the NUMBER_KEYS type. Therefore, it also falls within the PRINTABLE_KEYS range and, in turn, the ASCII_KEYS range. Like ALL_KEYS, these nine subtypes are also derived in ANSI_TERMINAL_SERVICES.

TABLE 2-4. ALL_KEYS SUBTYPES

SUBTYPE NAME	RANGE	KEYS
ASCII_KEYS	NUL .. DEL	128
CTRL_KEYS	NUL .. US	32
PRINTABLE_KEYS	SPC .. TILDE	95
UC_LETTERS	UC_A .. UC_Z	26
LC_LETTERS	LC_A .. LC_Z	26
NUMBER_KEYS	ZERO .. NINE	10
MAPPED_KEYS	DELETE .. FK100	114
EDIT_KEYS	DELETE .. DOWN_ARROW	14
FUNCTION_KEYS	FK1 .. FK100	100

2.2.4 Keyboard Bindings

Given any keyboard and KEYBOARD_INPUT's list of *logical keys*, a means must exist to link the two. Some *logical keys*, of course, are bound automatically because they correspond to members of the ASCII character set. After all, users should not need to map a printable ASCII character, because the standard dictates
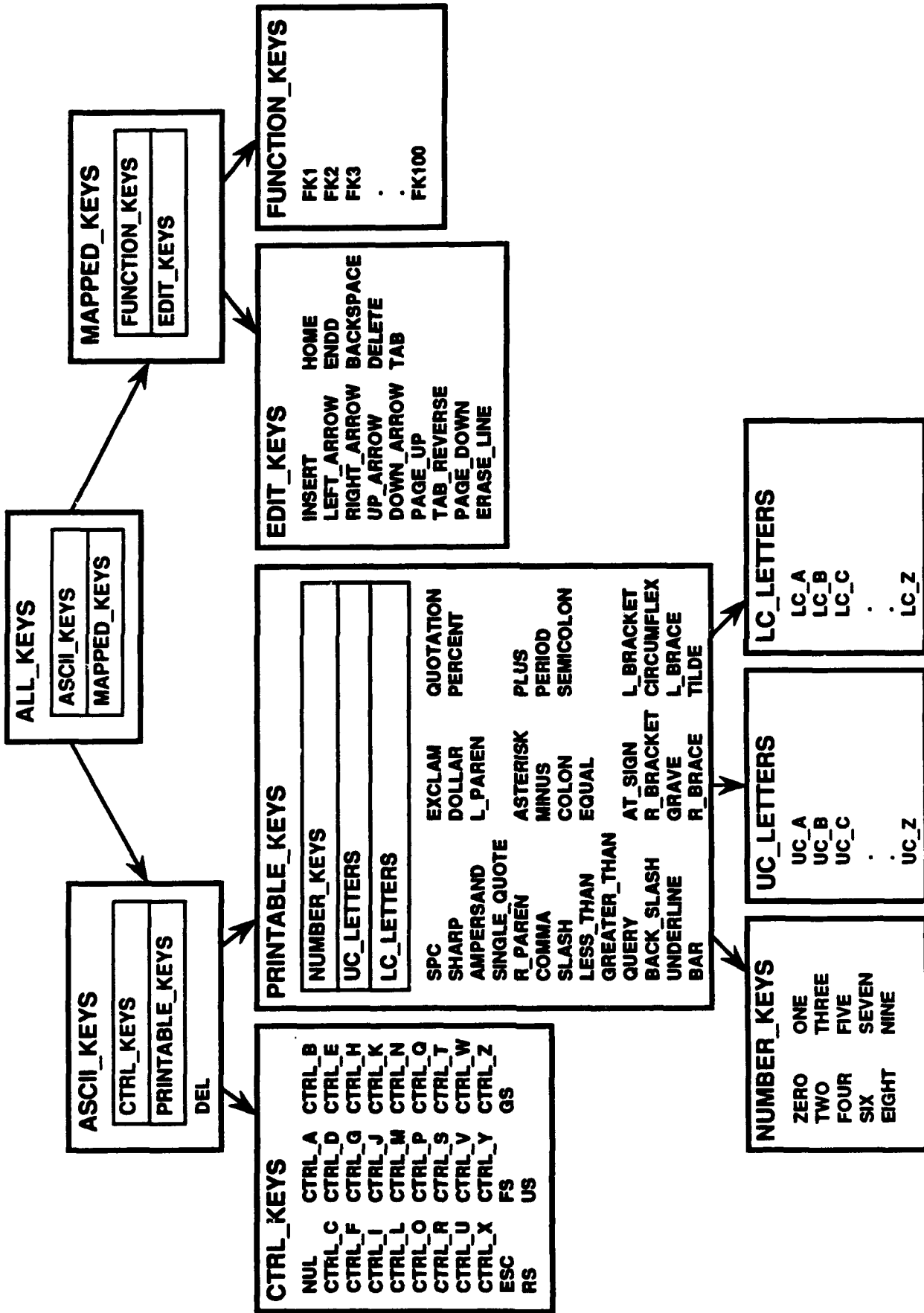
FIGURE 2-2. LOGICAL KEY HIERARCHY

commonality across keyboards. Pressing the spacebar, for instance, will generate an ASCII 32, regardless of a keyboard's *layout*. Therefore, *logical keys* corresponding to the 128 ASCII characters are automatically mapped to individual characters simply by their position within the enumerated type ALL_KEYS.

Non-ASCII *keystrokes*, however, conform to no standard and therefore must be explicitly mapped to their associated *escape sequence*. Of the 242 elements of ALL_KEYS, those of the MAPPED_KEYS subtype (2.2.3) must be bound in this manner. This range of *logical keys* includes those associated with editing keys and function keys. When one of these keys (e.g., an arrow key) is pressed, an *escape sequence* results, which often varies both in content and size among different systems.

Since ALL_KEYS was designed to accommodate any keyboard, it also allows physical keys with no *logical key* equivalent to be mapped to the function *logical keys* (FK1 - FK100). The *escape sequence* generated by a **Help** key, for example, could easily be mapped to FK1. Mappings also apply to the opposite case. That is, ALL_KEYS contains edit keys that some keyboards do not support. If this is the case, the functionality associated with the edit *logical key* could be mapped to an ASCII control character. In this manner, a keyboard with no **Home** key could associate the *keystroke* HOME with the **Control-H** character.

ANSI_TERMINAL_SERVICES records all of these necessary bindings with the MAP_LIST type. This type is an array of strings of type ESCAPE_SEQUENCE (3.4.1) indexed on the *logical keys* of the MAPPED_KEYS subtype (2.2.3). Depending on the *logical key* being mapped, the corresponding strings may be *escape sequences* or individual ASCII control characters. The function *logical keys* (FK1 -FK100), for example, may be mapped only to *escape sequences*. The other mappable *logical keys*, however, may be bound to *escape sequences*, control characters, or the ASCII DEL character. For instance, users may want to map TAB to **Control-I** (ASCII 9) and RIGHT_ARROW to <ESC>[C. Regardless of the *logical key*, the length of the strings being mapped is constrained by the MAX_ESC_LEN constant. The default length is five, but the constant may be set by the user in the constants file (see Section 2.8), thereby eliminating the need for recompilation.

Variables of the MAP_LIST type are assigned values at runtime by accessing a keyboard mapping file. When a terminal is being initialized by ANSI_TERMINAL_ SERVICES, the specified mapping file is read, and the individual keyboard is conveniently mapped to the functionality associated with the *logical keys*. The application writer merely sets up a mapping file, as described in Appendix B, and passes its filename as a parameter to CREATE_TERMINAL (2.4.1). (Appendix C lists a sample mapping file.) If a mappable *logical key* is not explicitly mapped in this file, its corresponding array element usually is assigned a string of blanks (ASCII 32). Since the mappings for the *logical keys* listed in Table 2-5 are common for many keyboards, however, they are explicitly initialized by KEYBOARD_INPUT as shown. Of course, users may overwrite any inappropriate defaults by specifying the correct mapping in the file.

TABLE 2-5. DEFAULT LOGICAL KEY MAPPINGS

LOGICAL KEY	DEFAULT MAPPING
TAB	Control-I (ASCII 9)
BACKSPACE	Control-H (ASCII 8)
DELETE	DEL (ASCII 127)
UP_ARROW	<ESC>[A
DOWN_ARROW	<ESC>[B
RIGHT_ARROW	<ESC>[C
LEFT_ARROW	<ESC>[D

By abstracting the keyboard as a flexible collection of *logical keys*, we have decoupled the functionality of a *keypress* from its actual representation. As a result, applications of ANSI_TERMINAL_SERVICES will be much more reusable. If these applications are consistently designed around the *logical keys*, rather than on the actual generated characters, then the application may be ported to different terminals without changing any code. Only the mapping file associated with the new keyboard will require modifications.

2.2.5   Screen Line Number

The LINE type from ANSI_TERMINAL_SERVICES is a subtype of the integers and is used to specify a line or row number for the display. A screen's lines are numbered beginning with one for the top line and continuing consecutively to the bottom line. The type is given a default range of 1 through 25. If the upperbound is too restrictive or too large, the MAX_LINE constant may be modified in the constants file (see Section 2.8).

2.2.6   Screen Column Number

Type COLUMN is derived in ANSI_TERMINAL_SERVICES to allow users to refer to individual columns on a display. These columns are numbered such that the left-most column is referred to as column one, and the right-most column corresponds to the MAX_COLUMN constant. The type assumes the default range of between 1

and 80. If this upper bound is inappropriate for the display in use, the MAX_ COLUMN constant may be changed in the constants file (see Section 2.8).

2.2.7 Cursor Screen Position

ANSI_TERMINAL_SERVICES provides the POSITION type to conveniently specify individual screen coordinates. This type is often used when repositioning the cursor, or when placing a string of text somewhere on the screen. This positioning is absolute in reference to the upper left-hand corner of the display. As shown in Table 2-6, POSITION is simply an encapsulation of the LINE (2.2.5) and COLUMN (2.2.6) types described above.

TABLE 2-6. CURSOR POSITION RECORD (TYPE POSITION)

COMPONENT	TYPE	VALUE RANGE	DEFAULT	REFERENCE
Row	LINE	1 .. MAX_RANGE	1	2.2.5
Column	COLUMN	1 .. MAX_COLUMN	1	2.2.6

2.2.8 Screen Column Offset

COLUMN_INCREMENT allows for a convenient way to specify a cursor's intended position within the current line. Unlike the positioning provided by COLUMN (2.2.6) or POSITION (2.2.7), that of COLUMN_INCREMENT is relative to the present cursor position. Because of this, advancing a cursor left or right within a line often involves a value of type COLUMN_INCREMENT. Another difference is that COLUMN_INCREMENT's range of values includes negative numbers, while COLUMN and POSITION values are always greater than zero. The negative range, of course, allows the cursor to move left towards column one. This type defines an integer range with a default of -80 to 80. To change this range (and consequentially the range of type COLUMN), users can modify the MAX_COLUMN constant in the constants file (see Section 2.8).

2.2.9 Supported Screen Colors

In support of color terminals, ANSI_TERMINAL_SERVICES allows users to specify the color of all screen I/O. These color settings may be changed repeatedly and at any time using the SCREEN_COLORS type. This enumerated type is

originally declared in ANSI_CURSOR_SERVICES and consists of the following eight color values:

- BLACK
- RED
- GREEN
- YELLOW
- BLUE
- MAGENTA
- CYAN
- WHITE

2.2.10   Text Intensity Settings

Many terminals (including those with monochrome displays) allow text to be written to the screen using varying brightness levels. When displaying text on the screen with ANSI_TERMINAL_SERVICES, the brightness of the resulting characters can be specified using the derived enumerated type INTENSITY_ SETTING. This type comprises the following two enumerals:

- DIM and
- BOLD

2.2.11   Text Blink Settings

To highlight important text, many terminals allow displayed characters to blink. Therefore, ANSI_TERMINAL_SERVICES provides the BLINK_SETTING enumerated type derived from the package ANSI_CURSOR_SERVICES. When writing text to the screen, applications may select a BLINK_SETTING of:

- BLINK or
- NO_BLINK

2.2.12   Cursor Appearance

When performing screen I/O, it is usually desirable to be able to vary the appearance of the resulting text. For instance, the foreground and background colors of text should be easily specified within a program. In addition to color settings, a cursor's appearance also includes its intensity or brightness level and its blink setting. The type TEXT_COLOR incorporates each of these settings into a single, convenient type. TEXT_COLOR is a record whose components are of types SCREEN_COLORS (2.2.9), INTENSITY_SETTING (2.2.10), and BLINK_SETTING (2.2.11), as described above (see Table 2-7).

TABLE 2-7. TEXT COLOR RECORD (TYPE TEXT_COLOR)

COMPONENT	TYPE	LEGAL VALUES	DEFAULT	REFERENCE
Background	SCREEN_COLORS	Black, Red, Green, Yellow, Blue, Magenta, Cyan, White	Black	2.2.9
Foreground	SCREEN_COLORS	Black, Red, Green, Yellow, Blue, Magenta, Cyan, White	White	2.2.9
Intensity	INTENSITY_SETTING	Dim, Bold	Dim	2.2.10
Blinking	BLINK_SETTING	Blink, No_Blink	No_Blink	2.2.11

2.2.13 Cursor Attributes

To fully describe the attributes of a cursor or its associated text, users must specify both screen location and appearance. When writing a string to a display, for instance, ANSI_TERMINAL_SERVICES must know where to place the text and how to set its color, intensity, and blink settings. ANSI_TERMINAL_SERVICES provides the ATTRIBUTE data type to encapsulate these parameters. As shown in Table 2-8, ATTRIBUTE's two components are of the types POSITION (2.2.7) and TEXT_COLOR (2.2.12), as described above.

TABLE 2-8. CURSOR ATTRIBUTE RECORD (TYPE ATTRIBUTE)

COMPONENT	TYPE	REFERENCE
Pos	POSITION	2.2.7
Color	TEXT_COLOR	2.2.12

2.2.14   Variable Length String

The type V_STRING is a discriminated record originally declared in package ANSI_CURSOR_SERVICES (see Table 2-9). Its discriminant is an integer of the STRING_SIZE range (3.2.1.1), which constrains the record's string component. This *variable-length string* allows users to assign *formatted strings* (i.e., text and its attributes) without regard to their individual lengths (see Section 2.2.15). The maximum size of the string is determined by the MAX_COLUMN constant and is assigned at runtime by the ANSI_CONSTANTS package (see Section 2.8).

TABLE 2-9. VARIABLE LENGTH STRING RECORD (TYPE V_STRING)

COMPONENT	TYPE	REFERENCE
Len (Discriminant)	STRING_SIZE	3.2.1.1
Str	STRING (1 .. LEN)	N/A

2.2.15   Formatted String

Type TEXT_REC is a record that groups a *variable-length string* of type V_STRING (2.2.14) with its ATTRIBUTE (2.2.13) values (see Table 2-10). This record is used for output services and allows the user to conveniently specify text and its intended screen position and appearance.

TABLE 2-10.   ORMATTED TEXT RECORD (TYPE TEXT_REC)

COMPONENT	TYPE	REFERENCE
Text	V_STRING	2.2.14
Att	ATTRIBUTE	2.2.13

2.2.16 String Layout Array

The LAYOUT type is an array of TEXT_RECs (2.2.15) and is intended to make writing multiple strings more convenient for the user. When several lines of text need to be displayed at one time, the user may use this array of *formatted strings* to group the write requests. Rather than issuing a write request for each line in a block of text, for instance, the user could initialize a LAYOUT array to hold the entire block. A single write request would then handle all the text at once. As illustrated later in Chapter 5 (Performance Characteristics), this approach is significantly more efficient.

The LAYOUT array is indexed by the LAYOUT_INDEX integer range. LAYOUT_INDEX constrains the array and, therefore, restricts the number of individual *formatted strings* that may be written with a single write request. This index range is determined at runtime when the ANSI_CONSTANTS package reads a value for the MAX_STRINGS_IN_LAYOUT constant (see Section 2.8).

2.3 PRIVATE DATA TYPES

In addition to deriving or redeclaring types from lower packages, ANSI_ TERMINAL_SERVICES also declares several private types of its own. These types provide the following capabilities required for a multitasking environment:

- Dynamic terminal attributes,
- Synchronization mechanisms for concurrent I/O requests, and
- An abstracted and protected terminal.

2.3.1 Dynamic Keyboard Bindings

The type KEY_MAP_ACCESS is private to ANSI_TERMINAL_SERVICES and is simply a pointer to an instance of the MAP_LIST (2.2.4) type. This access type allows a keyboard mapping list to be created dynamically. Since ANSI_TERMINAL_SERVICES is designed to support several terminals simultaneously, each terminal can allocate a variable of type KEY_MAP_ACCESS and obtain its own keyboard mapping list. Each terminal, then, has its own possibly unique keyboard bindings.

2.3.2 Dynamic Cursor Descriptor

ANSI_TERMINAL_SERVICES also defines the private CURSOR_ACCESS type. This type provides a pointer to an ATTRIBUTE (2.2.13) record. With CURSOR_ACCESS, a cursor descriptor may be allocated dynamically to record a cursor's attributes (e.g., position, appearance, etc.). Since ANSI_TERMINAL_

SERVICES supports an arbitrary number of terminals, it may dynamically allocate a cursor descriptor for each one.

2.3.3 Dynamic Screen Protection Task

ANSI_TERMINAL_SERVICES declares the private SCREEN_CONTROL task type to protect shared resources when output services are requested in a multitasking environment. Tasks rendezvous with a SCREEN_CONTROL server task in order to write to a screen, access the cursor attributes, or write to a port. However, ANSI_TERMINAL_SERVICES hides this tasking implementation from its applications. Instead, it defines several interface procedures in the nested package SCREEN (2.5). These procedures are identical in name and function to the entry points. (See Section 3.1.1 for a full discussion of this private monitor task.) By calling these procedures, users are guaranteed that an application's output requests will be handled in a safe, synchronized manner.

Since each terminal will need protection for its own resources when output services are requested, ANSI_TERMINAL_SERVICES also defines a type that allows an arbitrary number of these tasks to be created at runtime. SCREEN_ACCESS is private to ANSI_TERMINAL_SERVICES and is merely a pointer to the SCREEN_CONTROL task type described above. It is used internally to dynamically create a server task that provides protected output services to a terminal. Also, this dynamic type gives the keyboard task (2.3.4, below) access to the echoing services of its screen counterpart.

2.3.4 Dynamic Keyboard Protection Task

ANSI_TERMINAL_SERVICES defines the private KEYBD_CONTROL task type in order to control access to a keyboard's services. In effect, this is a monitor task through which other tasks access the shared cursor and port. For example, it contains entry points for line editing and reading from a port. (See Section 3.1.2 for a full discussion of this private task type.) Users of ANSI_TERMINAL_SERVICES, however, will not (and may not) rendezvous with the controlling task by calling its entry points directly. Instead, applications use the procedural interfaces defined in the nested KEYBD package described later (2.6). These convenient procedures correspond directly with the entry points of the KEYBD_CONTROL task type and ensure that all input service requests are accepted individually.

Since ANSI_TERMINAL_SERVICES was designed to support several terminals simultaneously, a monitor task is needed to synchronize access to each terminal's shared resources. ANSI_TERMINAL_SERVICES defines the KEYBD_ACCESS type to allow a keyboard protection task to be allocated dynamically as needed. This private type is simply a pointer to the KEYBD_CONTROL task type described above.

2.3.5   Terminal Manager

Inherent to using a terminal in a multitasking environment is the need to provide protection for its resources, namely the cursor and assigned port. ANSI_TERMINAL_SERVICES provides this protection with the private type TERMINAL. TERMINAL encapsulates the following:

- Terminal port identification
- Keyboard bindings associated with the terminal
- Terminal screen dimensions
- Current cursor attributes such as position and color
- Mutually exclusive access to input and output services

Since ANSI_TERMINAL_SERVICES was designed to support access to an arbitrary number of terminals from within a single application, most of TERMINAL's components are dynamically allocated. For instance, the cursor setting component is of type CURSOR_ACCESS (2.3.2). This is simply a pointer to an instance of an ATTRIBUTE record (2.2.13). Similarly, the keyboard map component is dynamic in nature. The components related to protecting the I/O are also access types (SCREEN_ACCESS (2.3.3) and KEYBD_ACCESS (2.3.4)). These are actually pointers to Ada tasks that provide I/O services in a manner that assures mutual exclusion in a multitasking environment. These underlying tasks synchronize I/O requests and prevent simultaneous access of the cursor or port by competing Ada tasks. The TERMINAL type effectively groups the shared data and monitoring tasks into a single record. The complete specification for type TERMINAL is shown in Table 2-11.

To properly use this type, applications will need a variable of type TERMINAL for each terminal being used. Each of these terminal resource managers is independent of the others and will provide the services listed above for exactly one terminal. That is, each individual terminal manager will have its own port identification, keyboard bindings, row and column constraints, protection mechanisms, and critical region for holding the cursor's attributes. The routines ANSI_TERMINAL_SERVICES provides to create and destroy such a manager are described in Section 2.4. As mentioned above, ANSI_TERMINAL_SERVICES also simplifies an application's interaction with the protection mechanisms by providing procedural interfaces to the protective tasks. These interface procedures comprise the nested packages SCREEN (2.5) and KEYBD (2.6), which are described later in this report.

TABLE 2-11. TERMINAL RESOURCE MANAGER RECORD (TYPE TERMINAL)

COMPONENT	TYPE	REFERENCE	ACCESSED TYPE	REFERENCE
Port	PORT_DATA	2.2.2	N/A	N/A
Num_Lines	LINE	2.2.5	N/A	N/A
Num_Cols	COLUMN	2.2.6	N/A	N/A
Key_Map	KEY_MAP_ACCESS	2.3.1	MAP_LIST	2.2.4
Cursor	CURSOR_ACCESS	2.3.2	ATTRIBUTE	2.2.13
Screen	SCREEN_ACCESS	2.3.3	SCREEN_CONTROL	2.3.3
Keybd	KEYBD_ACCESS	2.3.4	KEYBD_CONTROL	2.3.4

2.4 TERMINAL ALLOCATION AND DEALLOCATION

ANSI_TERMINAL_SERVICES allows applications to access an arbitrary number of terminals simultaneously. Each terminal, of course, requires a terminal resource manager to ensure the mutual exclusion necessary in a multitasking environment. Because of this, ANSI_TERMINAL_SERVICES provides the means for an application to dynamically create and subsequently deallocate terminal managers, as needed.

2.4.1 Terminal Initialization

ANSI_TERMINAL_SERVICES provides a convenient means of dynamically creating any number of terminal controllers. The CREATE_TERMINAL function localizes all necessary allocations and task instantiations. For example, CREATE_TERMINAL associates a keyboard mapping to the new terminal, assigns a port, records the screen's dimensions, and provides a protected abstraction of the terminal's cursor. Also at this time, the dynamic components of a TERMINAL (2.3.5) are allocated and assigned. This initialized TERMINAL variable is then returned to the caller. At this point, the caller has access to a terminal manager, providing safe I/O among multiple tasks. Before any screen I/O routines are called, however, the caller must initialize the shared cursor resource by calling the SET_CURSOR procedure described in Section 2.5.6.

2.4.2   Terminal Shutdown

When a terminal controller is no longer needed, a call to SHUTDOWN_ TERMINAL will remove the protection mechanisms set up for the terminal and deallocate the data types that provide this protection. This action terminates the TERMINAL (2.3.5) variable's task components, allowing graceful termination of an application program.

2.5   OUTPUT SERVICES

As discussed earlier, several difficulties arise when performing I/O in a multitasking environment. When writing to a port, care must be taken so that the output is not interleaved with that of concurrent I/O requests. Similarly, the cursor becomes a shared resource and must not be carelessly accessed when writing to the screen. The terminal resource manager defined by ANSI_TERMINAL_SERVICES incorporates routines and protection mechanisms designed to meet these and other needs. Specifically, ANSI_TERMINAL_SERVICES provides the following output services:

- Setting the cursor's positional and visual attributes
- Reading the cursor's current attributes
- Clearing areas of a display
- Writing asynchronously to a display
- Echoing user input
- Writing asynchronously to a port

The SCREEN package is nested within ANSI_TERMINAL_SERVICES and features the procedures of Table 2-12 to handle these output needs. Several of SCREEN's procedures (i.e., SET_CURSOR, SET_POSITION, and SET_COLOR) modify the cursor's positional and visual attributes. The cursor resource must be initialized in this manner prior to any screen I/O. These cursor routines are also useful for initializing the cursor for a subsequent read operation, because the input routines from the nested KEYBD package operate upon the current cursor attributes (see Section 2.6). In contrast, screen writes via the PUT procedure (2.5.10) do not use the current cursor settings. Rather, PUT requires cursor attributes as parameters. These settings are in effect only for the duration of the individual write operation. As the write completes, ANSI_TERMINAL_SERVICES resets the cursor to its state before the write call. This temporary cursor setting, therefore, has no effect upon subsequent writes.

2.5.1   Moving a Cursor via Absolute Screen Coordinates

The SET_POSITION procedure moves the terminal's cursor to the specified location on the screen. This new location is in terms of absolute screen coordinates in

TABLE 2-12. SUBROUTINES OF THE SCREEN SUBPACKAGE

PROCEDURE	REFERENCE	PROCEDURE	REFERENCE
SET_POSITION	2.5.1	GET_CURSOR	2.5.7
SET_COLUMN	2.5.2	CLEAR_TO_END_OF_LINE	2.5.8
GET_POSITION	2.5.3	CLEAR_SCREEN	2.5.9
SET_COLOR	2.5.4	PUT	2.5.10
GET_COLOR	2.5.5	PUT_STR	2.5.11
SET_CURSOR	2.5.6	PUT_PORT	2.5.12

relation to the upper left-hand corner of the display. Once set, the cursor's position remains fixed until another call explicitly repositions the cursor, or until a response to an interactive input service causes the cursor to move. SET_POSITION may be called any number of times to reposition the cursor as needed. Normally, this is done before each call to any of the input routines from the KEYBD package (2.6) because the input routines operate upon the current cursor attributes.

2.5.2   Moving a Cursor via Relative Column Increments

The SET_COLUMN procedure is included primarily to be used by the supporting LINE_EDITOR package (3.3) and allows the cursor to be repositioned within the current line in terms of a relative offset. For instance, a column offset of five will move the cursor to the right five spaces, whereas an offset of negative five will reposition the cursor to the left. The offset, of course, is relative to the cursor's current column number. The new position will remain in effect until another call is made to a SET procedure or until an input response moves the cursor.

2.5.3   Reading a Cursor's Screen Position

To read the current cursor position, ANSI_TERMINAL_SERVICES provides the GET_POSITION routine. The coordinates returned from this function are absolute screen coordinates of type POSITION (2.2.7).

2.5.4   Setting a Cursor's Appearance

SET_COLOR simply sets the cursor's appearance. That is, it resets the background and foreground colors, and the intensity and blink settings. This procedure may be called any number of times to reset the cursor as needed, and the settings remain in effect until the next call to SET_COLOR or SET_CURSOR (2.5.6). As described earlier, these cursor settings apply only to subsequent reads, because screen write requests require the cursor's write attributes to be specified as a parameter.

2.5.5   Reading a Cursor's Appearance

To read the current appearance settings assigned to a terminal's cursor, an application should call the GET_COLOR function. The returned ATTRIBUTE (2.2.13) record denotes the cursor's foreground and background colors, and the intensity and blink settings.

2.5.6   Setting a Cursor's Collective Attributes

When a user needs to reposition the cursor and change its appearance attributes, the SET_POSITION (2.5.1) and SET_COLOR (2.5.4) procedures may be called consecutively. Since this is undoubtedly a common operation, however, ANSI_TERMINAL_SERVICES provides the SET_CURSOR procedure to collectively set a cursor's positional and color attributes. SET_CURSOR can be called at any time to reposition and modify the terminal's cursor for future read operations. Also, SET_CURSOR should be called once before any screen I/O occurs, in order to properly initialize the shared cursor resource.

2.5.7   Reading a Cursor's Collective Attributes

The GET_CURSOR procedure replaces the need for consecutive calls to GET_POSITION (2.5.3) and GET_COLOR (2.5.5). A call to the function GET_CURSOR returns both the current position and appearance attributes of the terminal's cursor.

2.5.8   Clearing a Line of Text

CLEAR_TO_END_OF_LINE erases any text on the current line that is located to the right of the cursor. After this call, the cursor's position will remain unchanged.

2.5.9   Clearing a Screen

To erase an entire display at once, ANSI_TERMINAL_SERVICES provides the CLEAR_SCREEN procedure. In addition to clearing the display, this procedure also updates the shared cursor resource and positions the cursor in the upper left-hand corner of the screen, location (1, 1).

2.5.10   Writing to a Screen

ANSI_TERMINAL_SERVICES provides the overloaded PUT routine to allow multiple tasks to safely write to the same screen even in the presence of a reader task. This procedure is overloaded and displays either a single character, a string, or an array of strings. If several lines of text need to be written to the screen at one time, users should call the version of PUT that takes a LAYOUT (2.2.16) array of *formatted strings*. This method is both more convenient and more efficient than calling the string version of PUT to individually display each line in a block of text. In all cases, the actions necessary to ensure mutual exclusion are localized within the PUT procedure to simplify the interface with the application. The text will be displayed with the specified cursor attributes in a manner that does not adversely interfere with the cursor appearance or position associated with a concurrent read operation. Also, no other I/O request can interrupt the write or modify the cursor's attributes until the write completes. In this way, the cursor is protected, and many I/O operations can safely execute concurrently.

2.5.11   Echoing User Input

PUT_STR is a procedure that is used internally within ANSI_TERMINAL_SERVICES and its supporting package, LINE_EDITOR (3.3). It simply writes a string to a display and increments the cursor's column coordinate as specified. This routine, however, is not intended for users of ANSI_TERMINAL_SERVICES and provides no protection mechanisms when called outside of ANSI_TERMINAL_SERVICES or LINE_EDITOR. In fact, users are prevented from calling the procedure because one of its parameters is a private type hidden in a lower package. When writing to a display, therefore, users should always call the overloaded PUT procedures described above (2.5.10).

2.5.12   Writing to a Port

To write asynchronously to an assigned port, an application should call the overloaded PUT_PORT procedure. This routine sends a string or character directly to the port without fear of the message being jumbled with that of another I/O request. Although PUT_PORT is safe for direct port-to-port communication, users should normally not call it to write to a screen in a multitasking environment

because it provides no protection for the cursor resource. However, it may be used by sequential applications to produce scrolling outputs (see Section 1.2.7).

2.6   INPUT SERVICES

The input capabilities of ANSI_TERMINAL_SERVICES include the following:

- Interactive user prompts
- Timing mechanisms
- Port-based communication

Perhaps the most useful service of ANSI_TERMINAL_SERVICES is its ability to coordinate the activities of concurrent processes performing interactive I/O on a common 7-bit terminal. For example, writer tasks can easily update displayed information while a nonblocking reader task is prompting the user for input. As introduced in the last section, input routines use the current cursor settings, while screen output routines require attribute parameters. To set the cursor for a pending read, users must use the appropriate procedure (i.e., SET_CURSOR) from the SCREEN package (2.5). Because these reads are nonblocking, other Ada tasks may continue in the background. Because these reads are protected, the actions of writer tasks will not adversely interfere with the cursor or disrupt the display.

Although these reads are guaranteed not to block other tasks, it is possible for a read to never terminate. In situations where termination is critical, users should use the timed reads offered by ANSI_TERMINAL_SERVICES. These timed routines terminate the read if no response is detected within the specified time interval.

Similarly, direct port-to-port communication in a multitasking environment requires protection for the ports involved. When reading from a port, for instance, care must be taken to prevent jumbled data and ensure the integrity of the transmitted data. ANSI_TERMINAL_SERVICES provides this protection by coordinating all access to the shared port.

The above capabilities are provided by the nested package KEYBD. Although ANSI_TERMINAL_SERVICES incorporates private tasks to ensure the protection needed for such I/O (see Section 3.1.2), applications simply call procedures from the KEYBD package, which subsequently interact with the entry points of an underlying hidden task. Ada rendezvous still occur, of course, but their interface has been simplified and abstracted as the protected procedures listed in Table 2-13. Each of these procedures from KEYBD is described below.

TABLE 2-13. PROCEDURES FROM THE KEYBD SUBPACKAGE

PROCEDURE	REFERENCE
GET_KEY	2.6.1
GET_TIMED_KEY	2.6.2
GET_LINE_EDIT	2.6.3
GET_NOECHO	2.6.4
GET_TIMED_NOECHO	2.6.5

2.6.1 Reading a Keystroke

ANSI_TERMINAL_SERVICES provides the ability to safely read a single *keystroke* without blocking other tasks. It should be noted that a *keystroke* here refers to what we have previously referred to as a *logical key* (see Section 2.2.3). That is, a *keystroke* corresponds to an individual *keypress* and is independent of the ASCII characters resulting from that *keypress*.

The GET_KEY procedure will return the first *keystroke* encountered and will display a printable character if the programmer wishes. In case the programmer does not want the corresponding character displayed on the screen, the printing option may easily be turned *off*. For example, this latter scenario is well-suited for allowing a user to select a menu item. In either case, no carriage return is needed to complete the read; it terminates immediately after any key is pressed. If this key has not been mapped in the user's keyboard binding file, GET_KEY will return the NUL *logical key*.

2.6.2 Reading a Keystroke with Time Constraints

GET_TIMED_KEY provides the same functionality as GET_KEY above (2.6.1), but introduces a timing constraint. That is, it returns a representation of a single *keypress* and optionally displays a character on the screen. If no key is pressed within the specified time frame, however, the read will automatically terminate.

2.6.3   Line Editing

In addition to the above methods for reading individual *keystrokes*, ANSI_ TERMINAL_SERVICES also supports line editing in a multitasking environment. This line editor is invoked by a call to the GET_LINE_EDIT procedure within ANSI_ TERMINAL_SERVICES' nested package KEYBD. GET_LINE_EDIT offers several line editing options:

- Backspacing and deleting operations inherent to line editing
- In-line cursor movement
- Jumping to the beginning or end of the text
- Toggling between insert and overstrike mode
- Erasing a line with a single *keypress*
- Assignment of a default string to be accepted or edited
- Immediate termination upon detection of control and function keys

The above functionality is actually from the supporting package LINE_ EDITOR. Since it is accessed through ANSI_TERMINAL_SERVICES, however, it may be safely used in a multitasking environment. Any editing, for example, does not block background tasks, because of the synchronization and protection ANSI_ TERMINAL_SERVICES provides. Similarly, the output of concurrent tasks cannot disrupt the line editor.

Granted, all keyboards do not necessarily have keys labeled to match the above editing operations. Since ANSI_TERMINAL_SERVICES was designed to be highly reusable, however, this is not a problem. An easy method exists to map the above operators to any keyboard by means of a keyboard binding list (2.2.4), which is read from a file at runtime. As previously described, this file effectively binds any keyboard to the *logical keys* described previously (2.2.3). If a given keyboard does not have an **Erase Line** key, for example, no functionality is lost; the line erase operation could simply be mapped to an unused function key. In this manner, every editing operation listed in Table 2-14 is available for any keyboard with a sufficient number of assignable keys.

The string passed to the line editor is important in the following ways:

- Its length determines the size of the input field
- Its contents are displayed as the default input
- Upon return, it holds the contents of a user's response

If the string **Default** is passed to the line editor, for example, the user's response would be limited to seven characters, as dictated by the length of the parameter string. Regardless of whether the user accepts or edits the default, the input session ends when the user presses a key mapped to a control *logical key* (NUL .. US), a function *logical key* (FK1 .. FK100), or certain edit keys as listed in Table 2-14. In addition, this terminating *keystroke* is returned to the application along with the

TABLE 2-14. LINE EDITOR OPERATIONS

LOGICAL KEY	ACTION
INSERT	Toggle between insert and overstrike mode
LEFT_ARROW	Move cursor left one position
RIGHT_ARROW	Move cursor right one position
HOME	Mover cursor to beginning of input field
ENDD	Move cursor to end of input field
DELETE	Delete the character at the cursor, and shift characters to the right of the cursor
BACKSPACE	Delete the character left of the cursor, move the cursor left one position, and shift characters to the right of the cursor
ERASE_LINE	Erase the current input string, and move the cursor to the beginning of the input field
UP_ARROW DOWN_ARROW PAGE_UP PAGE_DOWN TAB TAB_REVERSE NUL .. US FK1 .. FK100	Terminate the line editor, and return the current string and terminating character

input sting. This is done in case an application needs to respond differently depending upon how the user ended the line editing session. For example, a concurrent Ada task could display a warning message on the screen as the user is using the line editor (see Figure 2-3). With a single *keypress*, the user could terminate the line editor and respond to the warning accordingly.

2.6.4 Reading Directly from a Port

The overloaded GET_NOECHO procedure in ANSI_TERMINAL_SERVICES provides a means of receiving a string or a single character from an assigned port. It includes the necessary protection mechanisms to prevent a read from receiving jumbled data. This procedure is overloaded so that it can return either a single ASCII character or a string of characters. Regardless of the parameter type, the read will

TIME: 10: 17: 34

*** WARNING : NEW LAUNCH DETECTED ***
PRESS: F1 for threat assessment
         F2 to defend
         F3 to edit again

EDIT SOME TEXT: Default text

Urgent warning and
Instructions from
separate Ada task

Operator may terminate
line editor and respond
to the warning by
pressing F1, F2, or F3.
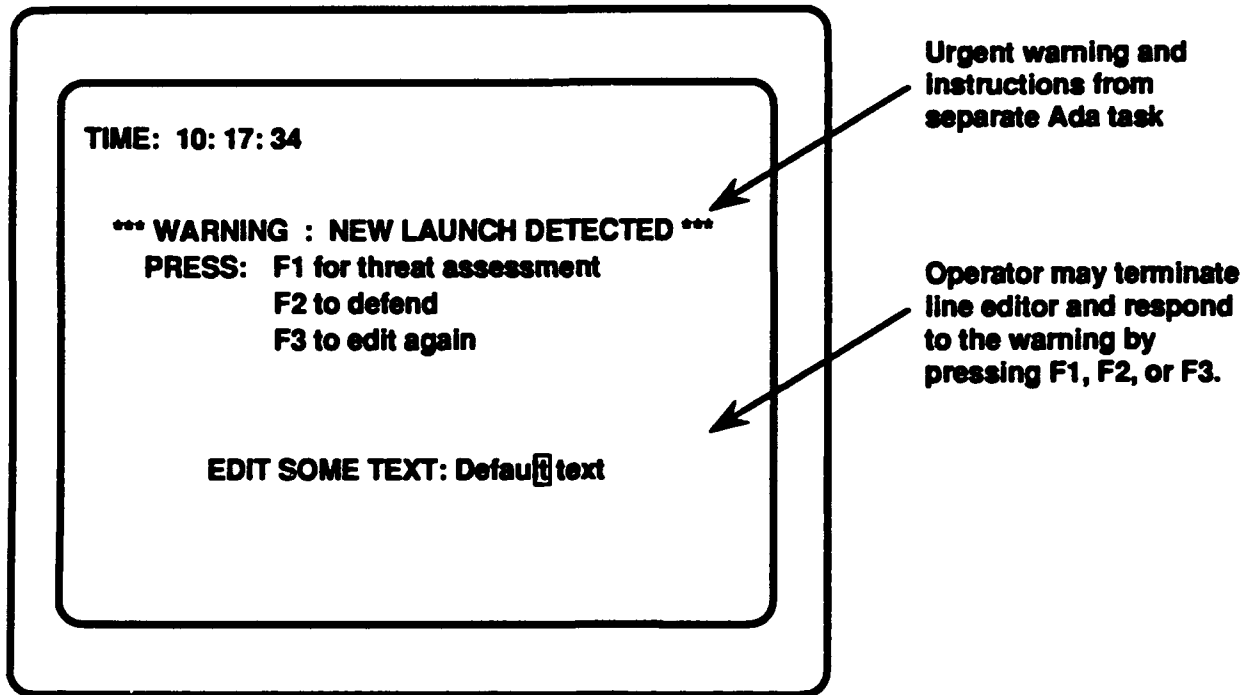
FIGURE 2-3. ONE USE OF THE LINE EDITOR TERMINATOR

terminate when either a carriage return is encountered, or the target string has been filled.

2.6.5   Reading Directly from a Port with Time Constraints

To avoid a nonterminating read request, users may elect to use the overloaded GET_TIMED_NOECHO procedure. The procedure offers the same functionality described above for GET_NOECHO (2.6.4) and adds a timing constraint. If a read requested via GET_TIMED_NOECHO does not terminate within the specified time, the read will automatically terminate and will return the data read before the timeout occurred. GET_TIMED_NOECHO is overloaded to handle input of either a single character or a string.

2.7   CONVENIENCE FUNCTIONS

ANSI_TERMINAL_SERVICES provides a number of functions designed to simplify its interface with application programs. Collectively, these routines provide the following services:

- Access to terminal data (e.g., port name, keyboard bindings, etc.)
- Converting between Ada strings and *variable-length strings*

2.7.1   Terminal Manager Data

Applications and ANSI_TERMINAL_SERVICES' lower packages often need to read components of the terminal manager record. This TERMINAL type (2.3.5), however, is private to ANSI_TERMINAL_SERVICES and is not directly accessible. To address this need, ANSI_TERMINAL_SERVICES provides the convenience functions of Table 2-15. Each returns a single component of the specified TERMINAL record.

TABLE 2-15. FUNCTIONS TO RETURN PRIVATE TERMINAL DATA

FUNCTION NAME	RETURNED TYPE	REFERENCE
THIS_TERM_PORT	PORT_DATA	2.2.1
THIS_TERM_MAP_LIST	MAP_LIST	2.2.4
THIS_TERM_NUM_LINES	LINE	2.2.5
THIS_TERM_NUM_COLS	COLUMN	2.2.6

2.7.2   Variable Length String Conversions

ANSI_TERMINAL_SERVICES provides two functions to aid in the assignment and conversion of *variable-length strings*. TO_V_STRING takes an Ada string of arbitrary length and converts it to a *variable-length string* of type V_STRING (2.2.14). In contrast, TO_STRING converts its V_STRING argument to a standard Ada string.

2.8 ELABORATED CONSTANTS

ANSI_TERMINAL_SERVICES and its supporting packages declare several constants commonly used to constrain data types such as strings and integer ranges. The values assigned to these constants, however, are not hard-coded. Instead, they are assigned values read from a file (**ANSI.DAT**) by the ANSI_CONSTANTS package. Because the file is read at package elaboration time, users may customize these settings without the need to recompile any source code. Appendix D describes the simple format of the constants file, and Appendix E shows the contents of the default file. Table 2-16 lists the constants initialized at runtime and the types they constrain. This table also relates whether a constant is visible to applications (i.e., declared in the specification of ANSI_TERMINAL_SERVICES) or merely used internally.

TABLE 2-16. IMPORTANT CONSTANTS INITIALIZED AT RUNTIME

CONSTANT NAME	DEFAULT VALUE	VISIBLE	CONSTRAINED TYPE	REFERENCE
MAX_PORT_NAME_SIZE	15	Yes	PORT_NAME	2.2.1
MAX_CHANNELS	16	No	CHANNEL_TYPE	3.5.1
MAX_ESC_LEN	5	No	ESCAPE_SEQUENCE	3.4.1
MAX_LINE	25	Yes	LINE	2.2.5
MAX_COLUMN	80	Yes	COLUMN	2.2.6
MAX_LONG_STRING_SIZE	1024	No	LONG_STRING_SIZE	3.2.1.2
MAX_STRINGS_IN_LAYOUT	30	Yes	LAYOUT_INDEX	2.2.16

2.9 EXCEPTION HANDLING

ANSI_TERMINAL_SERVICES declares a single exception (IO_ERROR) that is raised whenever a problem occurs within the package or its supporting packages. IO_ERROR is also raised if the user requests an impossible I/O service or if the configuration files (i.e., constants, keyboard mapping, etc.) are invalid. For example, IO_ERROR will be propagated to the caller if an attempt is made to reposition the cursor at position (1, 80) after creating a terminal with dimensions 24 x 40. Similarly, CREATE_TERMINAL will raise IO_ERROR if the keyboard bindings file does not follow the proper format, or if ASYNC_IO cannot open a channel, given the specified port name.

CHAPTER 3

DESIGN DESCRIPTION

Whereas the previous chapter described the specification of ANSI_TERMINAL_SERVICES and the complete programmer interface, Chapter 3 expands upon ANSI_TERMINAL_SERVICES' concurrency control and its five supporting packages. These topics are organized as follows:

- ANSI_TERMINAL_SERVICES
- ANSI_CURSOR_SERVICES
- LINE_EDITOR
- KEYBOARD_INPUT
- ASYNC_IO
- ANSI_CONSTANTS

3.1   ANSI_TERMINAL_SERVICES

As described in Chapter 2, ANSI_TERMINAL_SERVICES offers I/O services that are protected in a multitasking environment. Applications of ANSI_TERMINAL_SERVICES may easily incorporate multiple writer tasks that concurrently update a display even in the presence of a reader task. Although abstracted to the user as protected procedure calls to the SCREEN (2.5) and KEYBD (2.6) packages, these protection mechanisms are actually implemented via Ada task types. These monitor tasks, SCREEN_CONTROL (3.1.1) and KEYBD_CONTROL (3.1.2), are described here in Section 3.1.

The keyboard task essentially owns the cursor and is preempted by write requests that temporarily gain mutually exclusive access to the shared resource. That is, the input routines use the current cursor attributes, while all output services must *borrow* the right to access the cursor. Otherwise, writers may permanently change the cursor's appearance or send output to an area of the display where other tasks are concurrently performing I/O. Also, a writer must restore the cursor to its original state before control of the critical region is relinquished. This is required because the write may have preempted a concurrent read operation, and effectively *borrowed* the cursor and its present attributes. Since the write does not use the current cursor settings, the attributes must be specified as a parameter of the write request. These temporary attributes apply only to the write operation and are overwritten on completion of the write, as the cursor is restored to its original state.

In addition to servicing user output requests, the output control task must also address the needs of echoing user input. The concurrency control afforded by ANSI_TERMINAL_SERVICES and the interactions between the implementing tasks are illustrated in Figure 3-1.

Associated with each physical terminal controlled by an application program is a TERMINAL record (2.3.5). As described previously, this record contains terminal parameters, pointers to shared data regions, and addresses of the keyboard and screen tasks. As a terminal is created, relevant parameters and pointers are copied into each task body. For instance, the keyboard task records the address of the associated screen monitor task, so that it may safely echo user input. Both tasks, of course, require access to the shared cursor record. Since ANSI_TERMINAL_ SERVICES supports an arbitrary number of terminals, each must have its own monitoring tasks and shared data regions. Fortunately, the abstracted terminal manager hides the tasking implementation and dynamic shared cursor. As illustrated in Figure 3-2, an application program merely declares a TERMINAL variable (e.g., T1, T2, etc.) for each terminal it accesses.

3.1.1 Private Screen Monitor Task

ANSI_TERMINAL_SERVICES declares the private SCREEN_CONTROL task type to protect shared resources when output services are requested in a multitasking environment. Its entry points correspond to common output services required by both the user and the keyboard monitor task. The KEYBD_CONTROL task (3.1.2), for example, is dependent upon SCREEN_CONTROL to safely echo user input. Similarly, user tasks rendezvous with the screen monitor in order to write to a screen, set the cursor attributes, or write to a port. However, ANSI_TERMINAL_ SERVICES hides this tasking implementation from its applications. Instead, it defines several interface procedures in the nested package SCREEN (2.5). These procedures are identical in name and function to the entry points. For example, the SET_POSITION procedure (2.5.1) merely calls the SET_POSITION entry point (3.1.1.4) of the SCREEN_CONTROL task. By calling these procedures, users are guaranteed that an application's output requests will be handled in a safe, synchronized manner. The screen monitor task is documented below in the following sequence:

- Task design
- Local state data
- Local subroutine
- Task entry points

3.1.1.1 Task Design. SCREEN_CONTROL is a server task consisting of a startup entry and an endless loop with a selective wait. The selective wait includes a shutdown entry and twelve entries (one overloaded) devoted to output services (see Section 3.1.1.4). These output services include screen updates (e.g., asynchronous
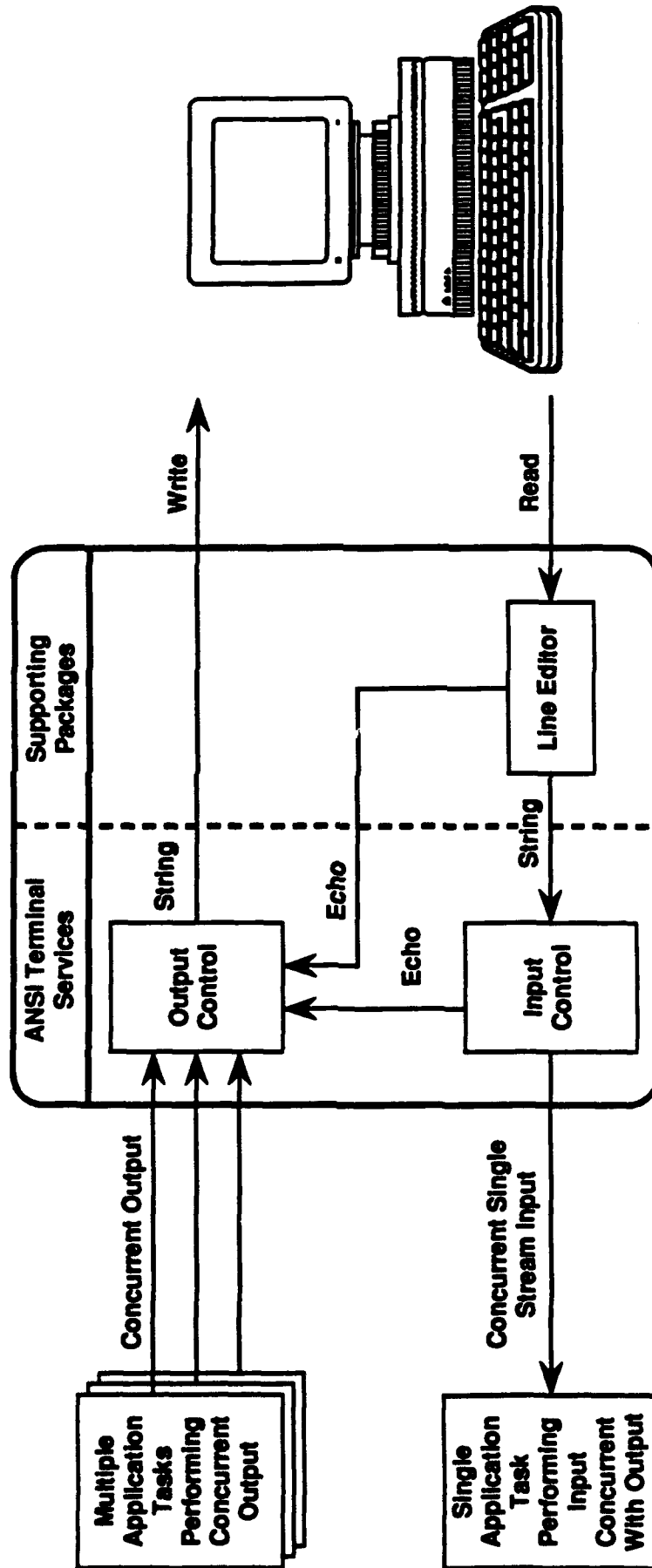
FIGURE 3-1. ANSI_TERMINAL_SERVICES CONCURRENCY CONTROL
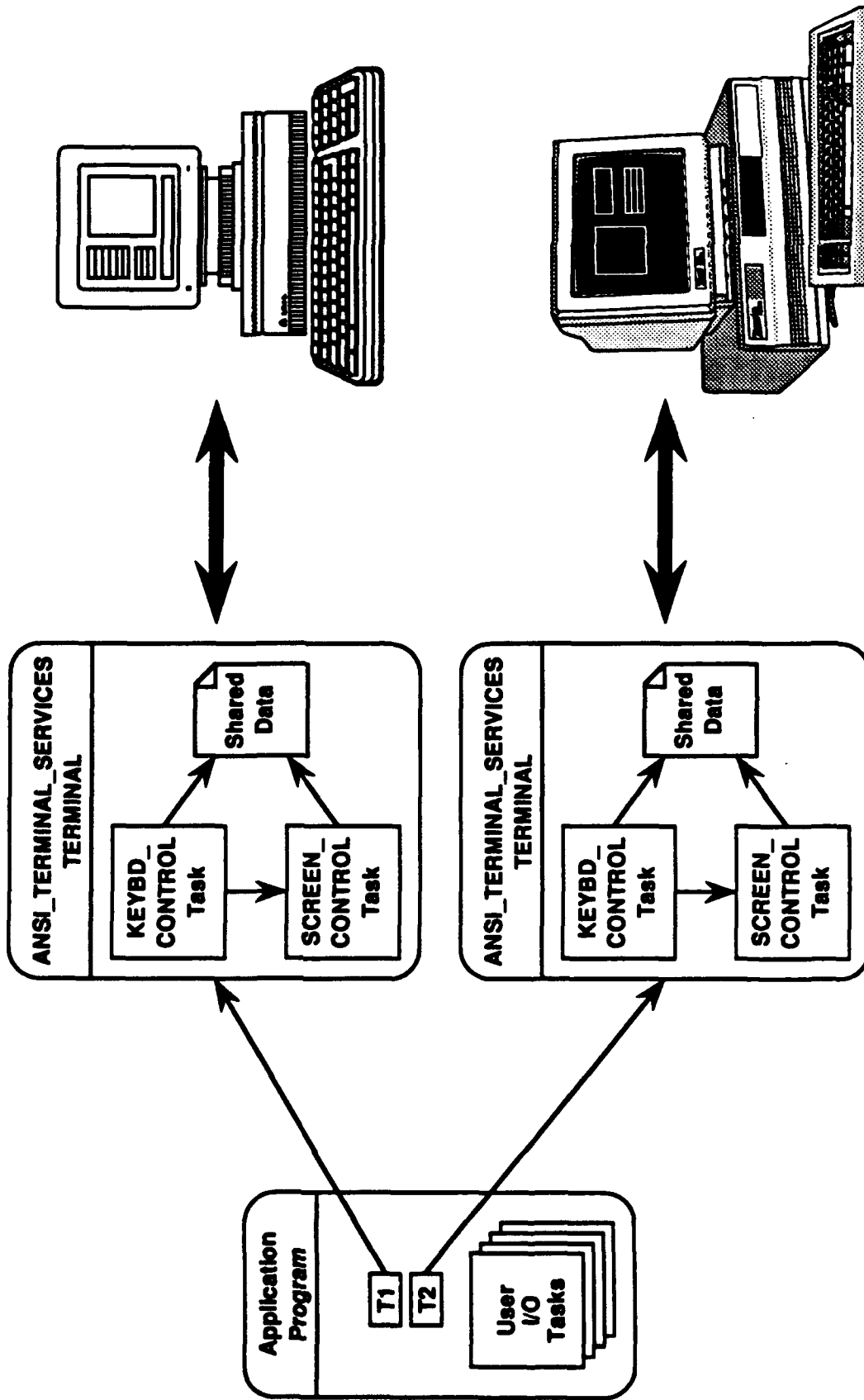
FIGURE 3-2. ABSTRACTED TERMINAL RESOURCE MANAGERS

writes, screen clearing, etc.) and cursor access (e.g., reading the cursor's screen position). A program design language (PDL) for this task is included in Figure 3-3. This figure also illustrates SCREEN_CONTROL's interaction with the keyboard monitor task and the line editor support package. That is, KEYBD_CONTROL (3.1.2) and LINE_EDITOR (3.3) must repeatedly use this task in order to safely echo user input.

As shown in the PDL, the SCREEN_CONTROL task type contains no exception handler; rather, its entries return an error flag indicating any problem encountered with an output service. The reason behind this involves a compiler-specific pragma (PASSIVE) described in Section 6.3. Briefly, this pragma is supported by the VERDIX Ada Development System (VADS) version 6.0 compiler and optimizes *passive tasks* such as SCREEN_CONTROL. To be eligible for this optimization, however, the task must contain no exception handler. SCREEN_CONTROL, therefore, sends error indicator flags back to its SCREEN (2.5) interface procedures, which in turn may raise an IO_ERROR exception.

This task type takes the system default for its Ada task priority. In the specification for ANSI_TERMINAL_SERVICES, the SCREEN_PRIORITY constant is commented out of the source code because of the system-dependent nature of Ada task priority values. Unlike other constants from ANSI_TERMINAL_SERVICES, this constant may not be assigned at runtime, because Ada's priority pragma requires a static expression. If the system default priority is not appropriate, users must, therefore, modify the specification for ANSI_TERMINAL_SERVICES by setting the constant value and removing the appropriate comment symbols.

3.1.1.2 Local State Data. SCREEN_CONTROL declares several variables needed by its entries. The most important of these record terminal parameters and provide access to the shared cursor resource. Other variables are useful during error checking and type conversions. The following list summarizes the variables local to the SCREEN_CONTROL task:

- THIS_TERM: This TERMINAL (2.3.5) variable is needed to give the screen task access to the attributes of the terminal it controls. For instance, it records the dimensions of the terminal as established by the user. Also, components of this record hold the terminal's port data and the address of the terminal's shared cursor resource.

- LENGTH: This POSITIVE variable records the length of the string argument to a write entry.

- REMAINING_SPACE: This INTEGER variable records the number of columns between the specified write coordinates and the rightmost screen column, as established by the user (see Section 2.3.5). If this value is less than the length of the argument string, the text will be truncated before the write.
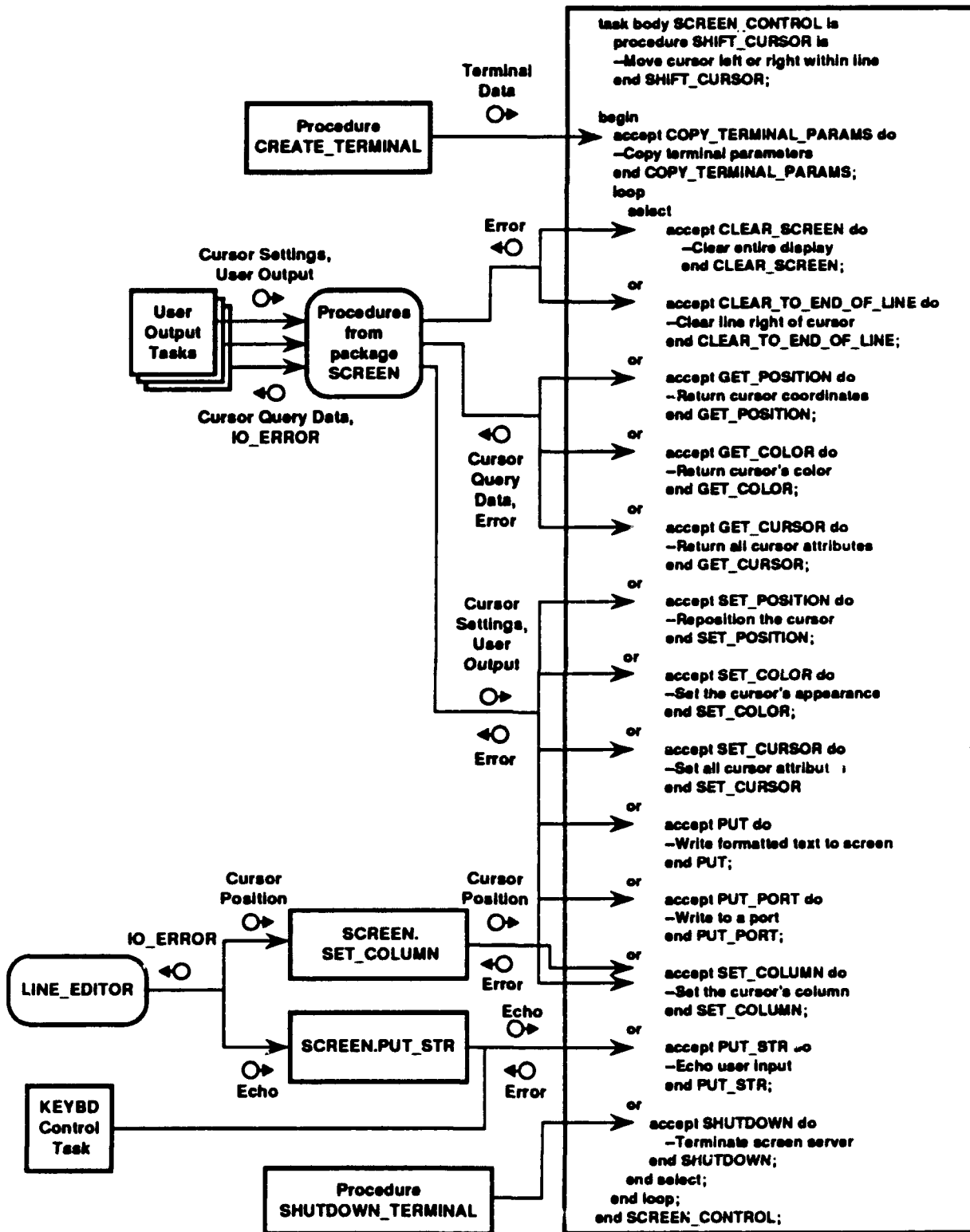
FIGURE 3-3. SCREEN CONTROL TASK

- TEMP_LAYOUT: This variable is an array of formatted text of type LAYOUT (2.2.16). It is used within the PUT (3.1.1.4) entry to convert between the derived LAYOUT type from ANSI_TERMINAL_SERVICES and the original type in ANSI_CURSOR_SERVICES. In order to accommodate any LAYOUT argument, TEMP_LAYOUT is constrained by the MAX_STRINGS_IN_LAYOUT constant (2.8).

- TEMP_PUT_STRING: This variable is a long *variable-length string* of ANSI_CURSOR_SERVICES' LONG_V_STRING data type (3.2.1.3). It comprises an arbitrary number of strings separated by the ANSI *control sequences* necessary to format each individual string. Because this string is used within the PUT protected screen output entry (3.1.1.4), it will also be appended with the *control sequences* needed to restore the cursor to its original state before the write. When the string is written, the given text will be displayed with the given attributes and will impose no side effects upon the shared cursor resource.

3.1.1.3 <u>Local Subroutine</u>. The SHIFT_CURSOR procedure is nested within the SCREEN_CONTROL task type. Given a value of type COLUMN_INCREMENT (2.2.8), this procedure updates the shared cursor resource and writes the *escape sequences* needed to reposition the cursor on the display. Because COLUMN_INCREMENT values may be positive or negative, this procedure may be used to shift the cursor right or left, respectively.

3.1.1.4 <u>Task Entry Points</u>. As discussed above (3.1.1), the entry points of this private task type are accessible to users only via the interface procedures of the SCREEN package. Because these interface routines involve record data types redeclared within ANSI_TERMINAL_SERVICES, the task entries must also perform necessary type conversions before calling routines from the lower-level packages (see Section 2.2). These task entries are described below:

- COPY_TERMINAL_PARAMS: This startup entry copies in terminal parameters, so that the SCREEN_CONTROL task has access to the shared cursor resource and the terminal settings (e.g., screen dimensions).

- CLEAR_SCREEN: This entry calls ANSI_CURSOR_SERVICES' CLEAR_ SCREEN procedure (3.2.2) to erase a display and reposition the cursor in the upper left-hand corner of the screen.

- CLEAR_TO_END_OF_LINE: This entry calls the CLEAR_TO_END_OF_ LINE procedure (3.2.2) from ANSI_CURSOR_SERVICES. As a result, the text to the right of the present cursor position is erased.

- SET_COLOR: This entry accesses the shared cursor data and updates its visual attribute settings (i.e., color, intensity, and blink settings). The *control sequences* needed to implement these new cursor settings are then

written to the display by calling ANSI_CURSOR_SERVICES' SET_
TEXT_COLOR procedure (3.2.2).

- GET_COLOR: This entry copies the cursor's current visual attributes into
  the given TEXT_COLOR (2.2.12) record.

- SET_POSITION: This entry updates the screen coordinates of the shared
  cursor record and writes the *escape sequences* necessary to physically
  reposition the cursor on the screen. This repositioning is accomplished by a
  call to ANSI_CURSOR_SERVICES' SET_CURSOR_POSITION procedure
  (3.2.2).

- SET_COLUMN: This entry adjusts the cursor resource's column setting and
  writes the appropriate *control sequences* to reposition the cursor on the
  screen. For example, the LINE_EDITOR support package calls this entry
  point (via the SCREEN.SET_COLUMN interface procedure (2.5.2)) to shift
  the cursor within the input field. Like SET_POSITION above, this
  repositioning is done by ANSI_CURSOR_SERVICES' SET_CURSOR_
  POSITION procedure (3.2.2). Also, error checking exists to ensure that the
  cursor is not adjusted beyond the established last column.

- GET_POSITION: This entry accesses the current coordinates of the shared
  cursor resource and copies them into the given POSITION (2.2.7) record.

- SET_CURSOR: This entry updates the shared cursor's collective attributes
  (i.e., position and appearance) based upon the specified ATTRIBUTE (2.2.13)
  values. The SET_ATTRIBUTE procedure (3.2.2) of ANSI_CURSOR_
  SERVICES is then called in order to write the *control sequences* required to
  display the new cursor settings.

- GET_CURSOR: This entry copies all current cursor attributes into the
  given ATTRIBUTE (2.2.13) record.

- PUT_PORT: This entry calls ASYNC_IO's PUT_ASYNC procedure (3.5.2.2)
  to write the specified string to the terminal's assigned port. Because no
  display is involved, the cursor resource is not updated.

- PUT_STR: This entry writes a string to a display via ASYNC_IO's
  PUT_ASYNC procedure (3.5.2.2) and updates the cursor position according
  to the specified column increment argument. This repositioning of the
  cursor is performed in a call to the SET_CURSOR_POSITION procedure
  (3.2.2) of ANSI_CURSOR_SERVICES. This write uses the current cursor
  settings and, therefore, is unsafe for user output tasks. Instead, this entry
  point is called only to echo user input within the associated KEYBD_
  CONTROL task. The LINE_EDITOR package also accesses this entry point

indirectly by calling the PUT_STR interface procedure (2.5.11) from the SCREEN package.

- PUT: This overloaded entry writes the specified formatted text to the screen, while protecting the cursor attributes. Once the text has been written, the proper *control sequences* ensure that the cursor is restored to its prerendezvous state. All output strings are checked and possibly truncated to prevent writing beyond the screen's established right border. One version of PUT handles a single string/attribute pair, while the second PUT writes an entire LAYOUT (2.2.16) array of *formatted strings*. For efficiency reasons, both are processed similarly and involve a call to ANSI_CURSOR_SERVICES' ASSEMBLE_LONG_STRING (3.2.2.2) procedure. This routine constructs a single string of the specified text, embedded with *control sequences* that provide the proper formatting. Since the *control sequences* necessary to restore the cursor to its original state are also appended to this string, a single call to ASYNC_IO's PUT_ASYNC procedure (3.5.2.2) safely writes the formatted text.

- SHUTDOWN: This entry terminates the screen control monitor task.

3.1.2 Private Keyboard Monitor Task

The KEYBD_CONTROL task type includes entries corresponding both in function and name to the input service procedures described previously in Section 2.6. Users of ANSI_TERMINAL_SERVICES, however, will not (and may not) rendezvous with this private controlling task by calling its entry points directly. As Figure 3-4 illustrates, applications instead route input service requests through the procedural interfaces defined in ANSI_TERMINAL_SERVICES' nested KEYBD package. When the GET_KEY procedure (2.6.1) is called, for instance, it in turn, calls the GET_KEY entry point (3.1.2.4) from KEYBD_CONTROL. The keyboard monitor task is described here as follows:

- Task design
- Local state variables
- Local subroutine
- Task entries

3.1.2.1 Task Design. The KEYBD_CONTROL task type is a server task comprised of an initialization entry and an infinite loop that contains a selective wait. This selective wait includes a shutdown entry and several others that handle input service requests individually by synchronizing calls to input routines from the supporting packages. Figure 3-4 includes a PDL for this task. Additionally, this figure shows the necessary interaction between this task and the terminal's associated screen monitor task. This dependency exists because of the need to intermittently echo user input. If a user inserts a character while line editing, for
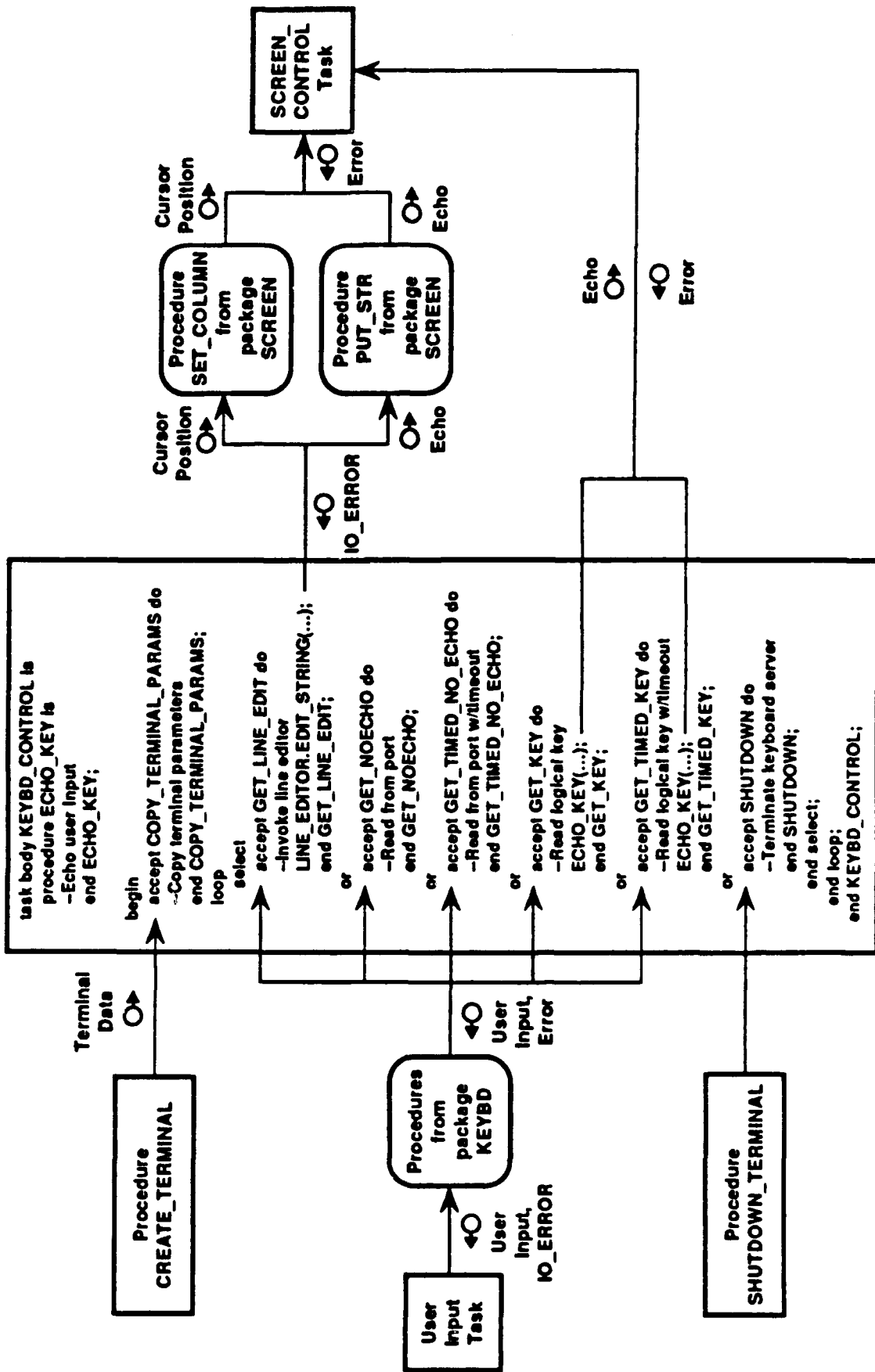
FIGURE 3-4. KEYBOARD CONTROL TASK

instance, the screen contents must be updated safely via a call to SCREEN_
CONTROL's PUT_STR entry (3.1.1.4).

As with SCREEN_CONTROL previously, the KEYBD_CONTROL task type
takes advantage of VADS 6.0's *passive task* optimization and must, therefore, use
error flags rather than exception handlers. If any problems occur with an input
service request, the error flag causes an exception (IO_ERROR) to be raised in the
KEYBD package (2.6) and subsequently propagated to the caller. For a full
discussion of this *passive task* optimization, see Section 6.3.

As with SCREEN_CONTROL above, this type takes its Ada task priority from
the system default (see Section 3.1.1.1). To change this, users must change the
KEYBD_PRIORITY constant in ANSI_TERMINAL_SERVICES and remove the
appropriate comment symbols.

3.1.2.2 Local State Data. The KEYBD_CONTROL task declares several local
variables required by its input services. Most importantly, one of these variables
provides the keyboard task with access to terminal parameters, shared resources, and
SCREEN_CONTROL's echo routines. The local state data is summarized below:

- THIS_TERM: This variable is of type TERMINAL (2.3.5) and is needed to
  copy in the attributes of the associated terminal. For example, THIS_TERM
  records the terminal's port data needed for subsequent input services. In
  addition, this record includes several dynamic components, allowing the
  keyboard control task to access the shared cursor resource, the user-defined
  keyboard bindings, and the SCREEN_CONTROL (3.1.1) entry point needed
  to echo user input.

- KEYSTROKE: This variable is of type ALL_KEYS (2.2.3) and records the
  logical key returned from a *keystroke* read.

3.1.2.3 Local Subroutine. Nested within the KEYBD_CONTROL task type is
the ECHO_KEY procedure. Given a *logical key*, this procedure converts it to a
printable character (if possible), displays the character at the current cursor position
with the current cursor visual attributes, and shifts the cursor right one position.
Like all output requests in a multitasking environment, this single-character write
must be properly synchronized to avoid side effects with concurrent I/O operations.
That is, the output request must be routed through the terminal's associated screen
protection task described earlier in Section 3.1.1. The ECHO_KEY procedure,
therefore, echoes the character by calling the PUT_STR entry point (3.1.1.4) of the
SCREEN_CONTROL task.

3.1.2.4 Task Entry Points. In addition to calling I/O routines provided by the
support packages, each KEYBD_CONTROL entry implements all necessary type
conversions so that users may access the functionality of lower packages using only
the visible derived types of ANSI_TERMINAL_SERVICES (see Section 2.2). As

described earlier (3.1.2), the entry points of this private task type are accessible to applications only through the identically named interface procedures of the KEYBD package. The task entries are listed and described below:

- COPY_TERMINAL_PARAMS: This startup entry copies the terminal parameters into the TERMINAL variable local to the KEYBD_CONTROL task. This gives the keyboard control task access to the shared cursor resource, terminal settings (e.g., keyboard bindings), and the screen monitor task needed to safely echo input.

- GET_LINE_EDIT: This entry invokes the line editing service described in Section 2.6.3 by calling LINE_EDITOR's EDIT_STRING procedure (3.3.2). The edit string is displayed at the current cursor position. Before the invocation, KEYBD_CONTROL ensures that the requested edit string does not pass the right border of the screen. If the parameter string is too long, it is truncated so that only that portion visible on the screen is passed to the EDIT_STRING procedure.

- GET_NOECHO: This entry reads a string of arbitrary length directly from a port by calling ASYNC_IO's GET_ASYNC_NOECHO (3.5.2.3). The read terminates when either the string is filled, or a carriage return is encountered. In either case, the number of characters successfully read is returned to the caller along with the input string.

- GET_TIMED_NOECHO: This entry is identical in functionality to the GET_NOECHO entry described above, with the addition of timing constraints. This timed read is provided by a call to GET_ASYNC_TIMED_NOECHO (3.5.2.4) from the ASYNC_IO package. If no input is detected within the given time interval, this procedure sets the timeout flag and terminates the read operation.

- GET_KEY: This entry reads a single *keypress* event by calling KEYBOARD_INPUT's overloaded GET_KEY procedure (3.4.2.2). This procedure traps and parses *escape sequences* and returns the single *logical key* bound to the pressed key. Additionally, this entry includes an echo flag. If the flag is set, the *logical key* resulting from the read is passed to the ECHO_KEY procedure (3.1.2.3) local to the keyboard task.

- GET_TIMED_KEY: This entry provides the same services as GET_KEY above, but places a timing constraint on the read. If no *keypress* event is detected within the given time interval, the read terminates, and the timeout flag is set. This timed read is performed by the timed version of KEYBOARD_INPUT's overloaded GET_KEY procedure (3.4.2.2).

- SHUTDOWN: This entry terminates the keyboard control monitor task.

3.2 ANSI_CURSOR_SERVICES

ANSI_CURSOR_SERVICES is responsible for writing ANSI *escape sequences* to a display in order to manipulate the cursor and control the screen's appearance. As mentioned in Section 1.2.3, these standard *escape sequences* are hard-coded and must be changed for nonstandard terminals. Most terminals, of course, adhere to the ANSI guidelines and recognize the *control sequences* used to modify the cursor's position and color. The remainder of this section is organized as follows:

- Visible data types
- Visible subroutines
- Local subroutines
- Constants

3.2.1 Visible Data Types

ANSI_CURSOR_SERVICES declares many data types related to cursor specification (i.e., screen position, color, etc.). Most of these types are derived or redeclared in ANSI_TERMINAL_SERVICES and were, therefore, described previously in Section 2.2 (see Table 2-2 from Section 2.2). Those types not interfaced in ANSI_TERMINAL_SERVICES are discussed here in Sections 3.2.1.1 through 3.2.1.4.

3.2.1.1 Variable-Length String Size. Type STRING_SIZE is an integer range that constrains the *variable-length string* type (V_STRING) described earlier in Section 2.2.14. STRING_SIZE defines a range between one and the constant MAX_COLUMN. A *variable-length string* is, therefore, confined to a maximum length equal to the user-defined maximum screen width. As discussed in Section 2.8, the MAX_COLUMN constant may be modified by the application writer without recompilation.

Some Ada compilers (e.g., Verdix Ada 6.0), however, require discriminated records to be constrained at compile-time. Since STRING_SIZE constrains the discriminated record V_STRING, and since STRING_SIZE is itself constrained by a constant (MAX_COLUMN) set at runtime, an upper bound has been placed upon it. This constraint is imposed by the STRING_SIZE_RANGE subtype and is hard-coded to a maximum length of 120. Similarly, MAX_COLUMN is constrained by the COLUMN_RANGE subtype and is limited to a maximum value of 120. Although these extra constraints are not required by all compilers (e.g., DEC Ada), they are employed here to increase ANSI_TERMINAL_SERVICES' compliance with multiple Ada environments.

3.2.1.2 Long Variable-Length String Size. As described earlier (2.2.16), ANSI_ TERMINAL_SERVICES allows users to specify several *formatted strings* that are to

be written to the screen collectively. ANSI_CURSOR_SERVICES provides a routine (ASSEMBLE_LONG_STRING, 3.2.2.2) that constructs a single *variable-length string* comprising an arbitrary number of individual *formatted strings*. This new string, therefore, must be large enough to hold the text and *escape sequences* necessary for formatting. Type LONG_STRING_SIZE addresses this need by defining an integer range that constrains the long *variable-length string* type LONG_V_STRING (3.2.1.3). Its maximum size is determined by the MAX_LONG_STRING_SIZE constant. As described in Section 2.8, this constant may be initialized at runtime.

As with STRING_SIZE above (3.2.1.1), the LONG_STRING_SIZE range requires an upper bound in order to comply with certain compilers (e.g., VADS 6.0). That is, some compilers require that a discriminated record's constraining range to itself be constrained at compile-time rather than at runtime. The additional subtype LONG_STRING_SIZE_RANGE is used here to meet this requirement. It statically constrains LONG_STRING_SIZE to be no longer than 1024.

3.2.1.3 Long Variable-Length String. Like type V_STRING described earlier (2.2.14), type LONG_V_STRING is a *variable-length string* implemented as a discriminated Ada record. Variables of type LONG_V_STRING, however, are allowed to be considerably longer, because they are used when converting an array of *formatted strings* into a single *variable-length string*. (As described earlier (2.2.16), this conversion is made in the interest of optimization.) The discriminant of LONG_V_STRING determines the length of the string and is constrained by the LONG_STRING_SIZE integer range described above (3.2.1.2) (see Table 31).

TABLE 3-1. LONG VARIABLE-LENGTH STRING RECORD (TYPE LONG_V_STRING)

COMPONENT	TYPE	REFERENCE
Len (Discriminant)	LONG_STRING_SIZE	3.2.1.2
Str	STRING (1 .. Len)	N/A

3.2.1.4 Hidden Column Increment. The private type PUT_STR_INCREMENT defines an integer range (-MAX_COLUMN .. MAX_COLUMN) used to specify the column offset when echoing user input. The type is private, because it is intended to prevent users of ANSI_TERMINAL_SERVICES from calling PUT_STR, a procedure intended to be used only internally to echo input. As discussed in Section 2.5.11, this private type allows only ANSI_TERMINAL_SERVICES and LINE_EDITOR to call the unprotected PUT_STR procedure.

3.2.2   Visible Subroutines

The functionality of many ANSI_CURSOR_SERVICES visible subroutines was previously described in Chapter 2. For example, the CLEAR_SCREEN procedure from ANSI_TERMINAL_SERVICES simply calls the CLEAR_SCREEN entry point of the screen task, which in turn calls the corresponding output procedure in ANSI_CURSOR_SERVICES. As shown in Table 3-2, users should refer to the earlier sections for descriptions of such output services.

TABLE 3-2. SUBROUTINES INTERFACED IN ANSI_TERMINAL_SERVICES

SUBROUTINE	REFERENCE	DESCRIPTION
CLEAR_SCREEN	2.5.9	Clears a display and positions the cursor in the upper left corner.
CLEAR_TO_ END_OF_LINE	2.5.8	Clears text to the right of the cursor within the same line.
SET_CURSOR_POSITION	2.5.1	Repositions the cursor.
SET_TEXT_COLOR	2.5.4	Assigns the cursor's visual attributes.
SET_ATTRIBUTE	2.5.6	Sets the cursor's positional and visual attributes.
TO_V_STRING	2.7.2	Converts an ADA string to a variable length string of type V_STRING.

Visible routines not interfaced in ANSI_TERMINAL_SERVICES are fully documented below. These routines are often needed by ANSI_TERMINAL_ SERVICES and provide the following capabilities:

• Disabling line wrapping
• Construction of optimized control strings
• Important type conversions

3.2.2.1 Disabling Line Wrap Mode. The procedure LINE_WRAP_OFF writes the ANSI *escape sequence* that explicitly disables line wrapping on the terminal's screen. In this way, output will be truncated at the rightmost column, so that no scrolling will occur. Otherwise, the contents of the entire screen would shift, and the positional values of the protected cursor resource would no longer accurately reflect

the contents of the actual screen. Also, an unwanted line wrap could cause a critical line of information to scroll off the top of the display.

     3.2.2.2 <u>Assembling a Screen *Layout* Control String</u>. The procedure named ASSEMBLE_LONG_STRING constructs a long *variable-length string* from an array of formatted text (See the PDL below). That is, it groups the text from a LAYOUT (2.2.16) variable into a single string and embeds the cursor *control sequences* needed to implement the formatting associated with each individual string. ASSEMBLE_ LONG_STRING is optimized so that it only embeds these *control sequences* when needed. If two successive strings are to be written with the same visual attributes, for example, the *control sequences* that set the cursor color are inserted only before the first string. Given the sluggish nature of screen I/O, it is important to reduce the number of written bytes in this manner.

     ASSEMBLE_LONG_STRING is called by ANSI_TERMINAL_SERVICES' overloaded PUT procedures (2.5.10). The current cursor settings are therefore passed to this procedure, because the cursor must be reset to its original state upon completion of the write request. Assembling and writing a single (possibly long) string is significantly more efficient than writing the specified formatting *control sequences*, writing the text, and then writing the *control sequences* needed to restore the cursor to its original state. This method is especially efficient when several *formatted strings* are to be written at once, as with the LAYOUT version of PUT. In this way, a single write can display an arbitrary number of *formatted strings*. This optimization reduces the number of interactions with the terminal device driver and avoids the associated temporal overhead.

```
procedure ASSEMBLE_LONG_STRING
    (LAYOUT_ARRAY : in      LAYOUT;
    OUTPUT_STR       :      out LONG_V_STRING ) is
begin

    -- Record original cursor attributes.

    for STRING_NUM in LAYOUT_ARRAY' range loop

        -- Embed escape sequences to reposition the cursor.

        if APPEARANCE_DIFFERENT_FROM_LAST then
            -- Embed escape sequences to change the cursor appearance.
            -- Record cursor appearance attributes.
        end if;

        -- Add actual text to be displayed on the screen.

    end loop;

    -- Add escape sequences to restore the cursor to original state.

end ASSEMBLE_LONG_STRING.
```

3.2.2.3 <u>Converting to a Private Column Increment</u>. PUT_STR_INCREMENT (3.2.1.4) is a private type intended to prevent application writers from calling the unprotected PUT_STR procedure used internally to echo user input. The routines from ANSI_TERMINAL_SERVICES and LINE_EDITOR that need to call PUT_STR, however, must be able to assign values to this private type. The convenience function TO_PUT_STR_INCREMENT allows for this by converting a value of the visible type COLUMN_INCREMENT (2.2.8) to the private type PUT_STR_INCREMENT.

3.2.2.4 <u>Converting from a Private Column Increment</u>. As described in the previous section, the ANSI_TERMINAL_SERVICES and LINE_EDITOR packages must be able to perform type conversions on the private PUT_STR_INCREMENT type (3.2.1.4). The convenience function TO_COLUMN_INCREMENT is used internally to convert a PUT_STR_INCREMENT value to a value of the visible COLUMN_INCREMENT type (2.2.8).

3.2.3   <u>Local Subroutines</u>

ANSI_CURSOR_SERVICES includes two local subroutines that generate the ANSI *escape sequences* necessary to perform the specified cursor assignment. For example, unique cursor control *escape sequences* are defined to reposition the cursor and modify its visual attributes. As discussed in Chapter 1, ANSI_TERMINAL_SERVICES only supports terminals that follow this standard. The standard *escape sequences* are, therefore, hard-coded within ANSI_CURSOR_SERVICES.

3.2.3.1 <u>Cursor Relocation Control Sequence</u>. Given screen coordinates of type POSITION (2.2.7), the REPOSITION_SEQUENCE function returns the standard control sequence necessary to reposition the cursor to the specified location. Because the size of this *escape sequence* will vary depending upon the specified screen location, the function returns a *variable-length string* of type V_STRING (2.2.14).

3.2.3.2 <u>Cursor Appearance Control Sequence</u>. The COLOR_SEQUENCE function returns the standard cursor control sequences needed to implement the cursor settings of its TEXT_COLOR (2.2.12) argument. The function returns a *variable-length string* of type V_STRING (2.2.14), which actually comprises several *escape sequences*. These sequences individually reset the cursor and set its color, blink, and intensity attributes.

3.2.4   <u>Constants</u>

Section 2.8 lists the constants used by ANSI_TERMINAL_SERVICES and its supporting packages. As discussed, these constants are assigned at package elaboration time by reading a constants file. Within ANSI_CURSOR_SERVICES, the following constants are initialized in this manner:

- MAX_LINE
- MAX_COLUMN
- MAX_LONG_STRING_SIZE
- MAX_STRINGS_IN_LAYOUT

3.3   LINE_EDITOR

The LINE_EDITOR supporting package defines a single visible procedure to implement the line editing service described in Section 2.6.3. This editor allows a user to move and edit freely within the input string and offers several operations mapped to the edit *logical keys* (e. g., HOME, ERASE_LINE, and INSERT). Editing of this nature requires explicit writes to update the contents of the current input field. For instance, inserting a character in the middle of an input string requires shifting text to the right of the cursor by one column. Because of this need to refresh the display with intermittent writes, and because LINE_EDITOR must allow for the presence of multiple I/O tasks, this package is dependent upon the protected writes provided by ANSI_TERMINAL_SERVICES' SCREEN_CONTROL monitor task (3.1.1). Specifically, EDIT_STRING and its nested subroutines repeatedly call the PUT_STR procedure (2.5.11) to properly and safely update the screen contents. This need for protected output within the input service results in the cyclic dependency between the ANSI_TERMINAL_SERVICES and LINE_EDITOR packages, as illustrated earlier in the package dependency diagram (Figure 2-1). This section describes the LINE_EDITOR package in the following order:

- Local data types
- Visible subroutines
- Local subroutines

3.3.1   Local Data Types

LINE_EDITOR defines no visible data types and only two local data types. These two types are described in the following two sections.

3.3.1.1 Insertion and Overstrike Mode Descriptor. When editing text, a *keypress* may either insert a character at the current cursor position, or overwrite the character under the cursor. MODES, a type local to LINE_EDITOR, is a type that encapsulates this line editing option. The enumerated type comprises the following two elements:

- INSERTT
- OVERSTRIKE

(Users should note the irregular spelling of the first enumeral; this is done to differentiate it from the INSERT *logical key* defined by the type ALL_KEYS (2.2.3).)

3.3.1.2 <u>Edit Keystroke Terminators</u>. As mentioned in Section 2.5.3, the line editing service terminates when the user presses a key mapped to a control *logical key* (NUL .. US), a function *logical key* (FK1 .. FK100), or certain edit keys, as listed previously in Table 2-14. LINE_EDITOR locally defines the EDIT_KEY_ TERMINATORS type to group those *logical keys* that cause the line editor to terminate. A subtype of type ALL_KEYS (2.2.3), EDIT_KEY_TERMINATORS consists of the six *logical keys* in the TAB .. DOWN_ARROW range (See Table 2-14).

3.3.2   <u>Visible Subroutines</u>

EDIT_STRING is the only visible subroutine defined in the LINE_EDITOR package. Users invoke this line editor by calling the GET_LINE_EDIT procedure (2.6.3) of the KEYBD package. The functionality of GET_STRING was previously described in Section 2.6.3. EDIT_STRING also encompasses several nested subroutines that are fully documented in the following section (3.3.3). These subroutines generally implement the primitive operations inherent to line editing, such as removing a character or checking for a full string. The following PDL describes this procedure.

```
procedure EDIT_STRING is
begin

   -- Echo user-specified default string
   while not DONE loop
     -- Get a keystroke
     if LOGICAL_KEY_IS_TERMINATOR then
       TERMINATOR_KEY := KEY;
       DONE := true;
     elsif KEY in PRINTABLE_KEYS then
       ADD_TO_STRING( CHAR_VAL( KEY ));
     elsif KEY in EDIT_KEYS then
       case KEY is
         when LEFT_ARROW = >
           -- Move cursor left one position
         when RIGHT_ARROW = >
           -- Move cursor right one position
         when INSERT = >
           -- Toggle between INSERTT and OVERSTRIKE modes
         when DELETE = >
           -- Remove character at the cursor
         when BACKSPACE = >
           -- Remove character to the left of the cursor
         when HOME = >
           -- Move cursor to beginning of the input string
```

```
            when ENDD = >
              -- Move cursor to the end of the input string
            when ERASE_LINE = >
              -- Erase the input string
          end case;
        end if;
      end loop;
    end EDIT_STRING;
```

3.3.3   Local Subroutines

This package defines several local subroutines necessary to implement a line editor. These routines are nested within the GET_STRING procedure (3.3.2) and provide the following capabilities:

- Testing for line editor termination
- Testing the status of the string (i.e., full or empty)
- Removing a character from the string
- Adding a character to the string (i.e., inserting or overstriking)
- Performing other primitive operations (e.g., erasing the input string)

3.3.3.1 <u>Determining if a Keystroke is a Terminator</u>. The Boolean function IS_ DONE_SEQUENCE determines if a specified *logical key* should cause the line editor to terminate. A TRUE value is returned if the *keystroke* is in the range of the control or function *logical keys*. Also, the function returns true if the *keystroke* falls within the EDIT_KEY_TERMINATORS (3.3.1.2) range defined locally within LINE_ EDITOR.

3.3.3.2 <u>Determining if an Edit String is Full</u>. The FULL_STRING function returns a Boolean value indicating whether the string being edited is currently full.

3.3.3.3 <u>Determining if an Edit String is Empty</u>. The EMPTY_STRING function returns a Boolean value indicating whether the string being edited is currently empty.

3.3.3.4 <u>Removing a Character</u>. The REMOVE_CHAR procedure deletes the character at the cursor. Characters to the right of the cursor are shifted left one position to adjust for the deletion. Since a character is being removed from the input string, a blank character (ASCII 32) is appended to the end of the string to maintain its proper length.

3.3.3.5 <u>Inserting a Character</u>. As its name implies, INSERT_CHAR adds a character to the input string at the current cursor position and shifts the cursor right one position. Text to the right of the cursor is also shifted right one position, so that no character is overwritten.

3.3.3.6 <u>Overwriting a Character</u>. The OVER_CHAR procedure overwrites the character under the cursor with the specified ASCII character. None of the remaining text is altered, and the cursor is shifted right one column.

3.3.3.7 <u>Adding a Character</u>. ADD_TO_STRING is a simple procedure consisting of only a single Ada CASE statement. Depending upon the current state of the line editor's MODES descriptor (3.3.1.1), the specified character is passed to another procedure. If the line editor is in INSERTT mode, procedure INSERT_CHAR (3.3.3.5) is called. Otherwise, the line editor is in OVERSTRIKE mode, and the character is passed to the OVER_CHAR procedure (3.3.3.6).

3.3.3.8 <u>Initiating an Edit Operation</u>. DO_EDIT_FUNCTION is called when an edit *keypress* is detected by the line editor. Depending upon the particular *logical key* encountered, appropriate action (i.e., a line editor primitive) is taken. If the INSERT *logical key* is pressed, for instance, the editor's MODES descriptor (3.3.1.1) toggles between INSERTT and OVERSTRIKE. This action and the functionality associated with each remaining edit *logical key* was previously listed in Table 2-14 of Section 2.6.3.

3.4 KEYBOARD_INPUT

KEYBOARD_INPUT is the support package responsible for declaring, mapping, and reading the *logical keys* described earlier (2.2.3, 2.2.4). These *logical keys* and their associated bindings effectively abstract the keyboard in an effort to enhance portability. By changing only the keyboard bindings file, an application that uses the *logical key* construct may be ported to another terminal without requiring costly maintenance and recompilation. The remainder of this section is outlined as follows:

- Visible data types
- Visible subroutines
- Local subroutines
- Constants

3.4.1 <u>Visible Data Types</u>

The most important data types KEYBOARD_INPUT defines are the ALL_KEYS and MAP_LIST types. These constructs allow users to configure an application at runtime in support of virtually any keyboard. ALL_KEYS (2.2.3), its subtypes (e.g., ASCII_KEYS, PRINTABLE_KEYS, etc.), and MAP_LIST (2.2.4) were fully documented in Chapter 2.

The only KEYBOARD_INPUT data type not derived in ANSI_TERMINAL_ SERVICES is ESCAPE_SEQUENCE. This type is simply a string constrained by the MAX_ESC_LEN constant. As discussed in Section 2.8, MAX_ESC_LEN is set at runtime and has a default value of five. If a keyboard generates *escape sequences* longer than five characters in length, users may easily modify the MAX_ESC_LEN value in the constants file.

3.4.2   Visible Subroutines

KEYBOARD_INPUT provides several subroutines described in Sections 3.4.2.1 through 3.4.2.5. Collectively, these routines offer the following capabilities for 7-bit terminals:

- Binding *logical keys* to their ASCII representation
- Reading a *logical key*
- Converting *logical keys* to other representations

3.4.2.1 <u>Reading a Keyboard Bindings File</u>. Section 2.2.4 described the MAP_ LIST array type. Indexed upon the mappable *logical keys*, this array is composed of strings of type ESCAPE_SEQUENCE (3.4.1). When a terminal is created, a keyboard mapping list is allocated and subsequently initialized by a call to KEYBOARD_INPUT's READ_KEY_MAPPINGS procedure. Given the name of a keyboard bindings file, READ_KEY_MAPPINGS initializes the array of *escape sequences* to values listed in the file. Initially, several of the most common mappings are assigned defaults before this file is read (See Table 2-5). If these defaults are inappropriate, of course, the user may specify different bindings in the mapping file. Appendix B lists the format for this keyboard mapping file, and Appendix C contains an example.

3.4.2.2 <u>Reading a Keystroke</u>. KEYBOARD_INPUT's overloaded GET_KEY procedures read a single *keystroke*, map the key (if necessary) and return the associated *logical key*. One version of the overloaded procedure incorporates a timing constraint; if no response is detected in the allotted time, GET_KEY sets a timeout flag and returns the NUL *logical key*. The full functionality of the overloaded GET_KEY procedure was previously described in Sections 2.6.1 and 2.6.2. However, one difference does exist between KEYBOARD_INPUT's GET_KEYs and the interface procedures from ANSI_TERMINAL_SERVICES; echoing is not optionally performed within KEYBOARD_INPUT. As with all output, echoing requires protection in a multitasking environment and is performed locally within ANSI_ TERMINAL_SERVICES. The following PDL summarizes GET_KEY:

```
procedure GET_KEY is
begin
```

```
if TIMED_VERSION then
    GET_SINGLE_KEYPRESS_TIMED_NOECHO;
else
    GET_SINGLE_KEYPRESS_NOECHO;
end if;

case FIRST_CHARACTER is
    when ESC = >
        if SECOND_CHARACTER = INITIALIZATION_CHARACTER then
            -- ESC key pressed
        else
            -- Find MAP_LIST entry matching this escape sequence

    when OTHER_CONTROL_CHAR | DEL = >
        -- Determine if edit keystroke is mapped to this character

    when others = >
        -- Printable key pressed
    end case;
end GET_KEY;
```

3.4.2.3 <u>Converting to a Printable Character</u>. The GET_CHAR_VAL function returns the single ASCII character associated with a printable *logical key*. For example, the function would return '$' if passed the DOLLAR *keystroke*.

3.4.2.4 <u>Converting to an ASCII Character</u>. Given a *logical key* in the ASCII_KEY range (2.2.3), GET_ASCII_VAL returns the ASCII position of the corresponding character. If given the UC_A *logical key*, for instance, GET_ASCII_VAL will return the integer 65.

3.4.2.5 <u>Converting to an Integer Equivalent</u>. GET_NUMBER_VAL is a function that converts a numerical *logical key* (ZERO .. NINE) to its integer equivalent.

3.4.3   Local Subroutines

Nested within the READ_KEY_MAPPINGS procedure are subroutines described in the following two sections.

3.4.3.1 <u>Initializing Keyboard Bindings</u>. Before reading a keyboard bindings file, READ_KEY_MAPPINGS calls the INITIALIZE_ESC_SEQS procedure. This sets the bindings of all mappable *logical keys* to blank (ASCII 32) characters. Some edit *keystrokes* are vital for the line editor service and are therefore given default bindings common to many terminals (see Table 2-5). These defaults, however, are

easily overwritten; if the *keystroke* is bound by the user in the mapping file, the default is discarded.

3.4.3.2 <u>Reading an Escape Sequence Binding</u>. If READ_KEY_MAPPINGS encounters an escape character symbol ('/E') when parsing the keyboard bindings file, the GET_ESC_SEQUENCE procedure is called to read the subsequent *escape sequence*. While reading the sequence, checks are made to ensure that it does not violate the size constraint set by the MAX_ESC_LEN constant (3.4.4).

3.4.4   Constants

KEYBOARD_INPUT defines only the MAX_ESC_LEN constant. This constant constrains the ESCAPE_SEQUENCE string type (3.4.1) and, in turn, the elements of the MAP_LIST array type (2.2.4). MAX_ESC_LEN defines the maximum length of *escape sequences* (including the ESC character) and has a default value of five. As discussed earlier in 2.8, the constant is set at runtime and may easily be configured by the user.

3.5   ASYNC_IO

ASYNC_IO is the lowest level I/O support package for ANSI_TERMINAL_ SERVICES. It performs channel assignments and asynchronous read and write operations by calling system service routines. Because it interfaces with operating system I/O routines, the body of ASYNC_IO is system-dependent. The specification, however, is host-independent and is documented here as follows:

- Visible data types
- Visible subroutines
- Constants

3.5.1   <u>Visible Data Types</u>

ASYNC_IO declares three visible data types related to channel assignment. PORT_NAME (2.2.1) and PORT_DATA (2.2.2) are derived in ANSI_ TERMINAL_SERVICES and were described earlier. The remaining type, CHANNEL_TYPE, is a private integer range constrained by the MAX_CHANNELS constant (3.5.3). In order to be as portable as possible, actual system-dependent channel identifications are recorded in an array in the package body. Type CHANNEL_TYPE serves as an index for this array. In this way, higher-level packages (including the application program) refer to this integer index, rather than the system-dependent channel identifier. Users of ANSI_TERMINAL_SERVICES, however, only need to specify a port name to the CREATE_TERMINAL (2.4.1)

routine; the CHANNEL_TYPE component of the associated PORT_DATA (2.2.2) is set by ASYNC_IO.

3.5.2   Visible Subroutines

ASYNC_IO declares several system-dependent procedures that perform low-level I/O services. Although the procedure bodies are system-dependent, the specifications of these subroutines are described in Sections 3.5.2.1 through 3.5.2.6. Specifically, these ASYNC_IO subroutines perform the following services:

- Channel initialization
- Asynchronous writes
- Asynchronous, nonblocking reads (optionally timed)
- *Escape sequence* detection and capture

As discussed in Section 1.0, ANSI_TERMINAL_SERVICES applications will likely perform input and output operations concurrently within several Ada tasks. Therefore, the I/O routines of this lowest package must be task synchronous/ process asynchronous to avoid having the input request block a user's entire process. Again, the implementation of this asynchrony is largely system-dependent. The version of ASYNC_IO in support of DEC's VMS operating system, for example, incorporates asynchronous system calls interfaced in DEC's TASKING_SERVICES package.

3.5.2.1 Initializing a Channel. INITIALIZE_CHANNEL provides access to a requested terminal port. Given the name of a port, INITIALIZE_CHANNEL calls the necessary system-dependent routine to assign an I/O channel. The resulting channel identifier is recorded within the package body's channel array, as discussed in 3.5.1. The index to this identifier, however, is recorded in the specified PORT_DATA record, along with the port name, and is returned to the caller.

3.5.2.2 Writing Asynchronously. PUT_ASYNC writes a specified string asynchronously to the given port. Because this is the only write in this collection of packages, the port assignment determines the target device. For example, this string could contain *escape sequences* destined to control a display or text being routed to another computer. In either case, PUT_ASYNC merely passes the string out the specified port; all protection mechanisms needed for concurrent I/O are provided by higher packages.

3.5.2.3 Stream Reading Without Echo. GET_ASYNC_NOECHO reads a stream of characters asynchronously from a port and explicitly disables screen echoing. Because this nonblocking read performs no formatting or *escape sequence* detection, an *escape sequence* is treated as a stream of individual characters. These characters (including the introductory escape character) are included in the returned string if space allows. If the string will not hold the complete *escape sequence*, the overflow characters will remain in the system's keyboard buffer until consumed by a

subsequent read. This read operation terminates either when a carriage return is encountered or when the user-specified string has been filled. In either case, GET_ASYNC_NOECHO also returns the number of characters read. This count does not include the carriage return terminator.

3.5.2.4 <u>Timed Stream Reading Without Echo</u>. GET_ASYNC_TIMED_ NOECHO performs the same services as in the preceding paragraph but adds a time constraint to the read. If no input is detected within the given time interval, GET_ASYNC_TIMED_NOECHO sets a timeout flag and returns the string and a counter indicating the number of characters successfully read. The timing implementation (e.g., interrupts, polling, etc.) is system dependent.

3.5.2.5 <u>Keypress Reading Without Echo</u>. GET_ASYNC_KEYPRESS_ NOECHO reads a single *keypress* event and traps *escape sequences* when they occur. As its name implies, this nonblocking read echoes no input to the display. The string associated with this procedure should be thought of as a single character, with an overflow area intended only for an *escape sequence*. If a user presses a printable key in response to this read, for example, the returned string will contain the detected ASCII character in position one, and the overflow area will remain unused. However, a key that generates an *escape sequence* may be pressed. In this case, GET_ ASYNC_KEYPRESS_NOECHO assigns the escape character (ASCII 27) to the first string position and fills the remainder of the string with the *escape sequence*. In either case, the read terminates immediately on any *keypress* and returns only the ASCII characters generated by this single *keypress* event.

3.5.2.6 <u>Timed Keypress Reading Without Echo</u>. Like the routine in the preceding paragraph, GET_ASYNC_KEYPRESS_TIMED_NOECHO reads a single *keypress* and traps *escape sequences*. This procedure also introduces a timing constraint. If no *keypress* is detected in the specified time interval, the read terminates and sets a timeout flag. The implementation of this timing mechanism (e.g., interrupts, polling, etc.) is system-dependent.

3.5.3   <u>Constants</u>

As discussed in Section 2.8, constants of ANSI_TERMINAL_SERVICES and its supporting packages are assigned at runtime via a constants file. Table 2-16 from that section lists these constants, their assigned value from the default constants file, and the data types they constrain. ASYNC_IO declares the following two constants that are initialized in this manner:

- MAX_PORT_NAME_SIZE
- MAX_CHANNELS

3.6 ANSI_CONSTANTS

The ANSI_CONSTANTS package is intended to allow users to easily customize ANSI_TERMINAL_SERVICES without recompiling any source code. At package elaboration time, this package reads the **ANSI.DAT** file and initializes its visible variables. As higher-level packages are elaborated, the constants of Table 2-16 are initialized with the values read from the constants file. (Pragma **ELABORATE** is used by higher-level packages to ensure that the ANSI_CONSTANTS variables are indeed assigned values before their associated constants are initialized.) In this way, constants and data types are constrained at runtime. The default constants file is listed in Appendix E. To customize ANSI_TERMINAL_SERVICES, application writers merely modify this file.

The remainder of Section 3.6, ANSI_CONSTANTS, is organized as follows:

- Visible variables
- Local constants
- Local state variables
- Elaboration code

3.6.1 Visible Variables

The specification of ANSI_CONSTANTS declares several variables from which higher-level packages initialize constants. These variables are all integers and are identical in name to the constants they subsequently define. Table 2-16 names these constants and lists their default values read from the visible variables of ANSI_CONSTANTS.

3.6.2 Local Constants

FILE_NAME is the constant string **ANSI.DAT**. At package elaboration time, this file is opened and read in order to assign ANSI_CONSTANTS' visible integer variables. The named initialization file must follow the format outlined in Appendix D.

3.6.3 Local State Variables

DATA_FILE is an Ada file descriptor of type TEXT_IO.FILE_TYPE and is associated with the constants initialization file. Defined in the package body, this file object is used in the elaboration code when reading the constants file.

3.6.4   Elaboration Code

When a compiled application program is run, the elaboration code of this low-level package is executed first. This sequence of statements opens the **ANSI.DAT** data file, reads an integer value for each of its visible variables, and closes the file.

CHAPTER 4

USAGE GUIDELINES

The program listing of Appendix F is a concise example of how to properly use ANSI_TERMINAL_SERVICES. Although it is not a very useful application, it does serve to demonstrate the capabilities and interface mechanisms of ANSI_ TERMINAL_SERVICES. Such a small application, of course, cannot fully demonstrate each of the services described in this paper. However, it does illustrate the following:

- Creation and deallocation of a terminal manager
- Protected asynchronous writes with varying cursor appearance attributes
- Optimized multiple writes using the LAYOUT type
- Line and screen clearing
- Setting of cursor attributes (e.g., position, appearance, etc.)
- Line editing service and use of its terminator
- Timed read of a *logical key* (without echo)
- System-independent reference to function keys

4.1   DESCRIPTION

This example illustrates a paneled screen introduced in Section 1.0. That is, the screen consists of several fixed regions into which the output from several concurrent tasks is mapped. At variable declaration time, the main procedure (EXAMPLE) creates a terminal manager by specifying a requested port name, the name of a keyboard binding file, and the dimensions of the associated display. In this example, the file of Appendix C is named as the mapping file. After a terminal manager is allocated, EXAMPLE initializes the cursor attributes, clears the associated display, and synchronizes the start of its embedded tasks. At that point, the main procedure simply goes into a busy wait until its tasks have terminated. A call to SHUTDOWN_TERMINAL (2.4.2) then deallocates the terminal manager and enables the program to terminate gracefully.

As illustrated in Figure 4-1, the EXAMPLE procedure incorporates three Ada tasks; two (WARNING, TIMER) are writer tasks, and the third (READER) repeatedly prompts the user for input. The READER task displays a prompt, repositions the cursor at the end of the prompt, and requests the line editor service. The associated string is given a default that an operator may edit or accept. When

TIME: 10: 17: 34

*** WARNING : NEW LAUNCH DETECTED ***
PRESS: F1 for threat assessment
        F2 to defend
        F3 to edit again

EDIT SOME TEXT: Default text

Concurrency
Control

EXAMPLE

Refresh

Delayed
Warning
Message
Task

Warn
User

Time of
Day
Periodic
Task

Shutdown

Alert Task

Input
Prompt
Task

Input, Threat Response
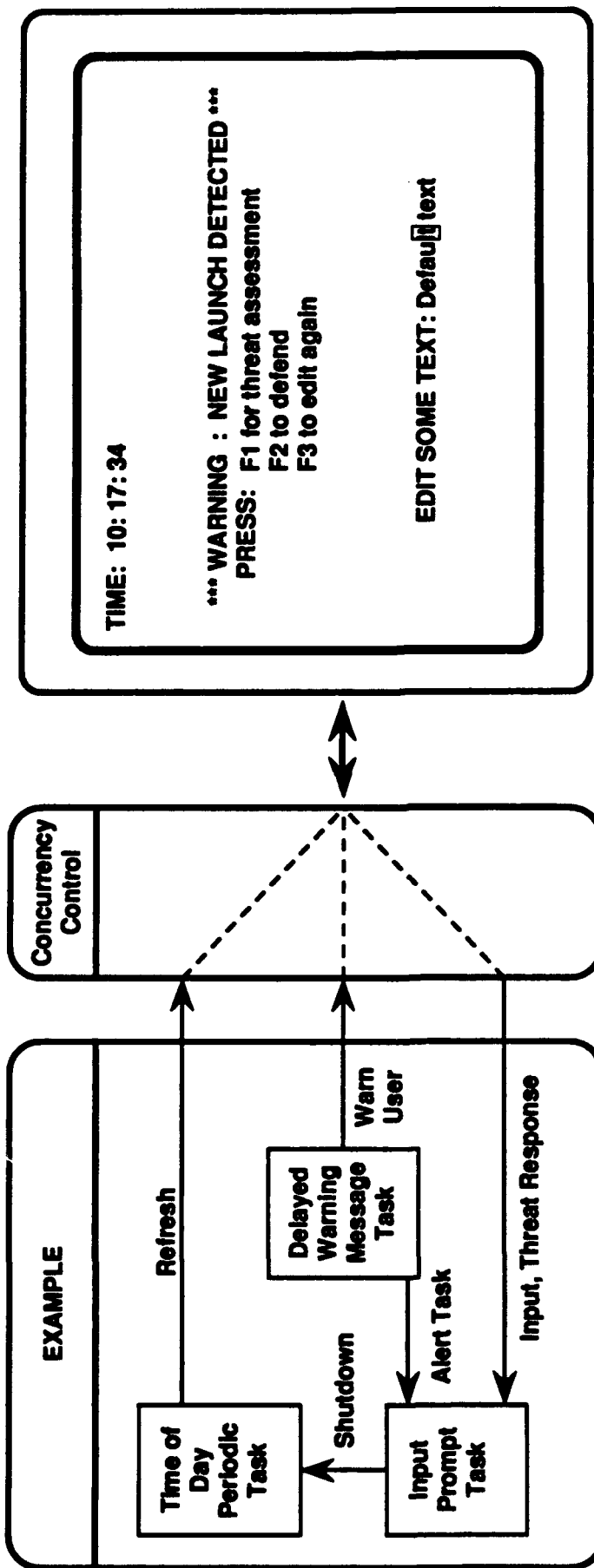
FIGURE 4-1. EXAMPLE APPLICATION

the operator is finished editing, the terminating *keystroke* is recorded in the variable TERMINATOR. Under normal conditions, the READER task will then loop and invoke the line editor again with the previous default string. (Task shutdown and the importance of this terminator will be discussed below.)

Meanwhile, the TIMER task methodically and concurrently updates a clock counter every second until requested by the READER task to stop. If no STOP entry call is encountered within one second, TIMER will increment its counters and call the ANSI_TERMINAL_SERVICES' PUT (2.5.10) procedure to display the results. Like the READER task, TIMER is associated with its own region of the display, into which no other tasks write.

Of EXAMPLE's three embedded tasks, WARNING is the simplest. It immediately delays for several seconds and then displays an urgent, flashing message. This message is made even more prominent by taking advantage of ANSI_TERMINAL_SERVICES' color support. Since the warning message consists of several strings with varying screen positions and attributes, the task sets a LAYOUT (2.2.16) array and calls the PUT procedure (2.5.10) to simultaneously write the strings. In this way, only a single PUT call is needed to write several lines to the screen. WARNING then informs the READER task of the threat by calling its SET_DANGER entry. After writing the warning message and setting READER's DANGER flag, the task terminates.

The DANGER flag is local to the READER task and indicates that a warning message has been displayed by another task. This message instructs the operator to select one of three choices, which have been mapped to the *logical keys* FK1, FK2, and FK3. By selecting FK3, the operator temporarily ignores the threat and invokes the line editor again. If the operator elects to counter the threat, however, the FK1 and FK2 options will terminate all callable tasks and display a message signifying that the appropriate action has been taken. At this point, the main procedure exits its busy wait and calls the SHUTDOWN_TERMINAL procedure (2.4.2) before terminating.

In addition to the different responses, the operator also has two methods for replying. The faster (and intended) method relies upon the line editor terminator mentioned earlier (see Section 2.6.3). If a user responds to the threat by pressing one of the response keys while editing, the current string will be accepted, and the appropriate action will be taken immediately. However, the user may choose to terminate the line editor with a carriage return or another nonresponse terminator key. The line editor accepts its string at this point, but the threat remains unhandled. To guarantee a threat response, the READER task then loops until the operator responds properly to the message. A timed read (GET_TIMED_KEY, (2.6.2)) is employed here, so that a bell may be sounded on each timeout.

In summary, procedure EXAMPLE and its Ada tasks demonstrate how ANSI_TERMINAL_SERVICES protects I/O resources in a multitasking

environment. By calling the proper routines, the actions of I/O operations produce no side effects to interfere with concurrent I/O actions. And because the interface to these protective services has been greatly simplified and localized to ANSI_ TERMINAL_SERVICES, the desired real time, paneled display of the EXAMPLE procedure is achieved quite easily.

4.2   POINT OF CONTACT

For further information or to obtain a copy of the software, contact:

Michael W. Masters
Naval Surface Warfare Center
Dahlgren Division
Code N35
Dahlgren, VA 22448

(703) 663-1611

mmaster@relay.nswc.navy.mil.

CHAPTER 5

PERFORMANCE CHARACTERISTICS

Like all software, the performance of ANSI_TERMINAL_SERVICES varies significantly with different hardware and compiler configurations. Even on common systems, observed performance may differ with varying resource allocations, system activity, and user privileges. Given the nondeterminism of software performance, however, meaningful insights may still be gained into these packages and into the efficiency of I/O in general.

Version 2.0 of ANSI_TERMINAL_SERVICES was initially developed with VAX/Ada on a DEC MicroVAX II under VMS and has recently been ported to a Sun Microsystem 4/260 workstation running the VERDIX Ada (VADS 6.0) compiler. Three individual tests were conducted to compare the two configurations. As expected, the performance of the Sun showed considerable improvements over that of the slower MicroVAX in all tests. After all, the Sun is a 10 MIP RISC machine, whereas the MicroVAX II is rated at only one MIP. In addition to the configuration comparisons, the tests also explore differences in the two methods of screen output ANSI_TERMINAL_SERVICES provides. Results of these tests may be compared to recognize the importance of the screen output optimizations explained earlier in Section 3.2.2.2. The benchmark results are given in Table 5-1 and explained below.

TABLE 5-1. BENCHMARK RESULTS

WRITE TEST	TIME (milliseconds)	
	MICROVAX II	SUN 4/260
String PUTs	335.0	150.7
Layout PUT	225.0	5.7

5.1   SCREEN OUTPUT

Writing formatted text to a display involves the following steps:

- Call to the string version of the overloaded PUT procedure
- Rendez/ous with the specified terminal's screen monitor task
- Validation of requested screen placement
- Adding formatting *control sequences* to the text
- Appending *control sequences* to restore the cursor to its original state
- Call to PUT_ASYNC to write the formatted text

In this manner, a single string of text is written with the given attributes, and the cursor state is not adversely affected. If several lines of text are to be displayed at the same time, the above steps may be repeated for each individual string. This method, however, is inherently redundant because *control sequences* are repeatedly written to set the cursor before and after each successive text string is written. A better method involves assigning a LAYOUT array (2.2.16) with the formatted strings and calling the *layout* version of PUT (2.5.10). This process involves the same steps above, except that the formatting text for all strings in the array is combined into a single string before the final cursor restoration sequences are appended. Thus, the desired display is produced with a single write, and the cursor is not unnecessarily restored to its original state after each individual string is written.

Two separate screen output tests were conducted on each configuration in order to compare the two output methods described above. Both tests displayed the four-line warning message of Figure 2-3 (2.6.3) and, therefore, produced the same output. The first method involved separately writing the four strings constituting the warning message, and required four calls to the string version of the overloaded PUT procedure (2.5.10). The second method, on the other hand, merely required a single call to the version of PUT in support of the LAYOUT (2.2.16) construct. As Table 2-12 shows, the second method is indeed optimized on both configurations to significantly accelerate the output of multiple *formatted strings*.

One potential reason for the performance difference concerns the resulting number of writes. While the first method involved four interactions with the underlying terminal device driver, the other method required only a single write. Similarly, the first method required four rendezvous with the screen monitor task. The majority of the observed difference, however, is attributed to the difference in the number of bytes each method required to achieve the desired results. As Table 5-2 illustrates, each of the four PUT calls from the first test generated 19 bytes to restore the cursor to its original state. The *layout* test, however, wrote a single long string and only needed to restore the cursor once. Also, the cursor color was not redundantly reset for each string of the array because the final three strings exhibit identical visual attributes. Although both methods produced the same results, the *layout* PUT required a single write and generated 89 fewer bytes of output. Given the sluggish nature of I/O, any elimination of unnecessary bytes is clearly desirable.

TABLE 5-2. BYTE COUNTS FOR SCREEN OUTPUT

BYTE COUNTS	MULTIPLE STRING PUTs					LAYOUT PUT
	1	2	3	4	Total	
Formatting Sequences	27	23	23	23	96	64
Text	40	31	19	23	113	113
Cursor Restoration	19	19	19	19	76	19
Total Bytes Written					285	196

Writers of real-time applications, however, may still not be satisfied with the output performance exhibited above. For example, the VAX results show that an optimized *layout* write to display *four formatted strings* required 225 milliseconds. Because the CPU would be out of the control of other tasks during this write, a 225 millisecond delay may prove to be dangerously long for real-time systems. For example, other important real-time tasks may be refused immediate execution while an earlier screen write request is being serviced. To meet such real-time deadlines, designers may choose to implement a screen write with multiple string PUTs rather than using the more efficient *layout* PUT. Thus designers may have to compromise between maximum *efficiency* and maximum *responsiveness*. Unfortunately, these goals may often conflict with one another.

CHAPTER 6

FUTURE WORK

Although ANSI_TERMINAL_SERVICES and its supporting packages provide an effective and convenient method for performing terminal I/O concurrently among Ada tasks, there are several areas in which the set of packages could be enhanced. Future upgrades would likely involve the following capabilities:

- X Window System support
- Decoupling the line editor and ANSI_TERMINAL_SERVICES
- *Passive task* support
- User-defined line editor terminators
- Enhanced support for monochrome displays
- Asynchronous transfer of control
- UNIX signal handling I/O
- Ports to additional operating systems

6.1   X WINDOW SYSTEM

Given the growing popularity and availability of the X Window System, it seems appropriate for ANSI_TERMINAL_SERVICES to offer compatibility. For example, a window could simulate an individual display. Although difficult, the current packages could possibly be supplemented with X Windows routines that access the X Windows *event queue*. Applications could then call the standard or X Windows routines, as needed. Alternatively, one or more of the packages could be made generic and instantiated with routines in support of X Windows or standard terminals. A less daunting and more probable solution, however, would involve a completely separate set of packages exclusively in support of the X Window System.

6.2   STAND-ALONE LINE EDITOR

Unfortunately, a cyclic dependency exists between the LINE_EDITOR and ANSI_TERMINAL_SERVICES packages (see Figure 2-1). The body of ANSI_TERMINAL_SERVICES is dependent upon LINE_EDITOR, of course, in order to offer the line editing service. However, the editor also requires ANSI_TERMINAL_SERVICES to provide necessary screen echoing. Editing of this nature often involves intermittent writes to update the contents of the input field. If a character in the

middle of the field is deleted, for example, characters to the right of the cursor must be shifted. Because of the need for protected I/O in a multitasking environment, these updating writes must be properly synchronized by ANSI_TERMINAL_SERVICES; hence, the cycle.

Because of this awkward dependency structure, the LINE_EDITOR package cannot exist independently of ANSI_TERMINAL_SERVICES as a stand-alone editor package. This is inconsistent with the other supporting packages; each of these may be used independent of ANSI_TERMINAL_SERVICES. (Of course, this is only applicable when multiple tasks are not performing I/O.) If the line editor were a *generic* package, however, this problem would likely be solved. The argument to the instantiation would simply be output routines responsible for echoing characters within the input field. ANSI_TERMINAL_SERVICES would instantiate the editor with routines that echoed to the screen safely in a multitasking environment (see PUT_STR, (2.5.11) and SET_COLUMN (2.5.2)). If an application is performing I/O and is not using ANSI_TERMINAL_SERVICES (and therefore involves no tasking), it could instantiate the line editor with ASYNC_IO's PUT_ASYNC routine and ANSI_CURSOR_SERVICES's SET_CURSOR_POSITION. In this manner, LINE_EDITOR could be decoupled from ANSI_TERMINAL_SERVICES, and yet remain accessible to both tasking and nontasking applications.

6.3   PASSIVE TASKS

The syntax and functionality of the Ada tasking model takes many forms, or idioms. Unfortunately, an intertask rendezvous requires an expensive operating system context switch for most of these forms. Some compilers, however, optimize certain task idioms in an effort to avoid the overhead associated with context switching. Some monitor (or server) tasks, for example, are well suited to such optimization. Containing no independent active thread of execution, these tasks merely service others when invoked and are appropriately termed *passive tasks*.

The VADS version 6.0 compiler supports this optimization with the PASSIVE pragma. This pragma results in the conversion of a *passive task's* entry calls into procedure calls protected by runtime system semaphores. (See the VADS 6.0 reference manual for a full discussion of pragma PASSIVE.) The resulting optimized calls are much more efficient than Ada's traditional rendezvous mechanism because it eliminates the context switch. Although not widely supported among vendors, *passive task* efficiency will soon be provided by the Ada 9X *protected record* construct.

Because of the heavy overhead involved with the standard Ada rendezvous, ANSI_TERMINAL_SERVICES would benefit greatly if its passive monitor tasks were optimized in the above manner. These tasks have been specifically designed to conform to the *passive task* idiom. However, even the VADS *passive task* implementation places certain constraints on their use. Among these is the limit that *passive tasks* cannot be used as record components or as the product of an

allocator operation. Thus this optimization of ANSI_TERMINAL_SERVICES must await removal of *passive task* constraints by Verdix.

6.4 USER-DEFINED TERMINATORS

Presently, the list of *logical keys* that terminate a line editing session is hard-coded into the LINE_EDITOR package. Specifically, the line editor will terminate when a function or control *logical key* is encountered. Also, those edit *logical keys* that have no corresponding line editor operation (e.g., PAGE_UP, TAB, etc.) cause the editing session to end. As a future enhancement, the user could select which *logical keys* will act as line editor terminators. The application could take the default terminators listed above or could specify a subset. If a user did not want a control *keystroke* to terminate the editor, for example, any such *keystrokes* would simply be ignored.

6.5 MONOCHROME SUPPORT

Although ANSI_TERMINAL_SERVICES presently supports both color and monochrome displays, it is admittedly more aligned to color support. For example, users must specify foreground and background colors whenever the cursor attributes are changed or passed as parameters. Since the colors (BLACK and WHITE) never change for a monochrome screen, such explicit color selection seems redundant. Also, ANSI_TERMINAL_SERVICES writes *escape sequences* to change the cursor colors. These *escape sequences* have no effect on a monochrome display and are, therefore, unnecessary and inefficient.

To better support monochrome displays, a flag could be included in the terminal manager record that indicates the capabilities of the associated terminal. If a monochrome monitor is being used, for instance, the unnecessary *escape sequences* could be avoided when the cursor's attributes change. Perhaps several procedures could also be overloaded so that their call lines eliminate the need to specify the color attributes of the cursor. Additionally, ANSI_TERMINAL_SERVICES could support reverse video for monochrome screens in an effort to simulate a second color.

6.6 ASYNCHRONOUS TRANSFER OF CONTROL

As currently implemented, a read operation may only be terminated by the user. The example outlined in Chapter 4 illustrates this scenario because its line edit operation terminates only when the user concludes the input normally cr presses the F3 terminator key. Such a situation, however, may be undesirable and potentially dangerous in a real-time environment. While one application task is in the process of reading data, for instance, another task may need to intervene and terminate the

read. ANSI_TERMINAL_SERVICES should allow for this possibility. Ada 9X will easily solve this problem with its *selective entry call*.

We could presently offer the same functionality without Ada 9X by dynamically creating and destroying reader tasks. However, this has a potentially serious flaw. Some implementations of the Ada runtime do not reclaim all memory associated with dynamically-created tasks when they become terminated. VADS 6.0, for instance, does not reclaim task control blocks. The resultant memory leakage can be lethal for a program that should theoretically run forever.

6.7   SIGNAL HANDLING I/O FOR UNIX

Presently, the UNIX version of ASYNC_IO's body implements nonblocking I/O by systematically *polling* the input channel for available characters. As discussed earlier in Section 1.2.2, *polling* is costly and is prone to unwanted delays. Furthermore, the optimal *polling* interval may be difficult to find and may easily differ between vendors and even among a vendor's platforms. Therefore, performance of the UNIX release of ANSI_TERMINAL_SERVICES would undoubtedly benefit by converting its nonblocking I/O routines to an interrupt-driven (i.e., signals) implementation.

6.8   SUPPORT FOR ADDITIONAL OPERATING SYSTEMS

Reusability was a major goal in the design of ANSI_TERMINAL_SERVICES and its supporting packages. In fact, only the body of the lowest package (ASYNC_IO) is system-dependent. Two versions of this package presently exist. One implements VMS system services in support of DEC's VAX/VMS operating system. Another version includes system calls to the UNIX operating system. Future enhancements could include additional versions for other operating systems, such as Microsoft Corporation's MS-DOS.

BIBLIOGRAPHY

Booch, G., *Software Engineering With ADA*, The Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, 1987.

"Escape and Control Sequences," *VMS I/O User's Reference Manual: Part 1*, Section 8.1.2.4, Digital Equipment Corporation, Maynard, MA, June 1990.

Gehani, N., *ADA: Concurrent Programming*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1984.

Installing and Using the VT320 Video Terminal, Digital Equipment Corporation, Maynard, MA, 1987.

Masters, M. and Kuchinski, M., *Software Design Prototyping Using ADA*, NSWCDD TR-82/417, Sep 1983, NSWCDD, Dahlgren, VA.

Thimbleby, H., "The Design of a Terminal Independent Package," *Software-Practice and Experience*, Vol 17(5), May 1987, pp 351-367.

VADS 6.0 Programmer's Guide for Sun-4 SunOS, Section 2.2, Verdix Corporation, Chantilly, VA, Apr 1989.

APPENDIX A

ANSI_TERMINAL_SERVICES SPECIFICATION

```
-------------------------------------------------------------------------
--
--              Dahlgren Division, Naval Surface Warfare Center
--
--
-- Authors:              M. Masters
--                       C. Chalkley
--
-- Department:           Combat System Technologies Branch (N35)
--
-- Revision History:
--    1/31/92            C. Chalkley
--       - Version 2.0 completed.
--
--
-------------------------------------------------------------------------
```

```
-------------------------------------------------------------------------
-- ANSI_TERMINAL_SERVICES v2.0
-------------------------------------------------------------------------
-- Purpose:
--    This package offers protected asynchonous terminal I/O among multiple
--    Ada tasks.  Specifically, ANSI_TERMINAL_SERVICES defines a terminal manager
--    which provides the synchronization necessary to perform I/O in a multi-
--    tasking environment.  This package also provides the complete interface
--    users will need to access the functionality of all supporting pacakges.
--
-- Effects:
--    - The expected usage is:
--      1. Create or modify the constants initialization text file, "ANSI.DAT".
--      2. Call CREATE_TERMINAL to allocate and initialize a terminal manager
--         for each terminal the application controls.
--      3. Call SET_CURSOR to initialize the application's terminal display.
--      4. Call the subroutines of the nested KEYBD and SCREEN packages
--         for I/O services.
--      5. Call SHUTDOWN_TERMINAL to deallocate each terminal manager.
--    - A valid keyboard mapping text file must be specified when a terminal
--      manager is created via CREATE_TERMINAL.
--    - The exception IO_ERROR is raised when any problems arise.
--    - Two tasks (types KEYBD_CONTROL & SCREEN_CONTROL) are activated for
--      each terminal created.
--    - Suggested renaming:  package ANSI renames ANSI_TERMINAL_SERVICES;
--    - A user's application is expected to be dependent only upon this package
--      and should not WITH the supporting packages.
--    - For full written documentation, see NAVSWC/TR-91/783, "Ada Based
--      Multitasking Terminal I/O"
--
-- Performance:
--    These routines of version 2.0 have been optimized for time by eliminating
--    unnecessary device driver interactions and by reducing the number of bytes
--    needed to format text for a terminal screen.  The embedded monitor tasks
--    also take advantage of compilers supporting passive task optimization.
-------------------------------------------------------------------------
```

```
with SYSTEM;
with ASYNC_IO;
with KEYBOARD_INPUT;
with ANSI_CURSOR_SERVICES;
with ANSI_CONSTANTS;

pragma ELABORATE( ANSI_CONSTANTS, ANSI_CURSOR_SERVICES );
```

```
package ANSI_TERMINAL_SERVICES is

   package CURSOR renames ANSI_CURSOR_SERVICES;

   MAX_PORT_NAME_SIZE : constant INTEGER := ANSI_CONSTANTS.MAX_PORT_NAME_SIZE;

   -- Maximum dimensions of terminal screens
   MAX_LINE   : constant INTEGER := ANSI_CONSTANTS.MAX_LINE;
   MAX_COLUMN : constant INTEGER := ANSI_CONSTANTS.MAX_COLUMN;

   -- Maximum number of formatted strings in a layout array
   MAX_STRINGS_IN_LAYOUT : constant INTEGER :=
       ANSI_CONSTANTS.MAX_STRINGS_IN_LAYOUT;


   --------------------------------------------------------------------------
   -- The following types are derived so that users of ANSI_TERMINAL_SERVICES
   -- will not have to WITH lower packages.
   --------------------------------------------------------------------------

   type PORT_NAME is new ASYNC_IO.PORT_NAME;
   type PORT_DATA is new ASYNC_IO.PORT_DATA;    -- terminal port descriptor

   type ALL_KEYS           is new KEYBOARD_INPUT.ALL_KEYS;    -- logical keys
   subtype ASCII_KEYS      is ALL_KEYS range NUL  .. DEL;
   subtype CTRL_KEYS       is ALL_KEYS range NUL  .. US;
   subtype PRINTABLE_KEYS  is ALL_KEYS range SPC  .. TILDE;
   subtype UC_LETTERS      is ALL_KEYS range UC_A .. UC_Z;
   subtype LC_LETTERS      is ALL_KEYS range LC_A .. LC_Z;
   subtype NUMBER_KEYS     is ALL_KEYS range ZERO .. NINE;
   subtype EDIT_KEYS       is ALL_KEYS range DELETE  .. DOWN_ARROW;
   subtype FUNCTION_KEYS   is ALL_KEYS range FK1     .. FK100;
   subtype MAPPED_KEYS     is ALL_KEYS range DELETE  .. FK100;

   type MAP_LIST is new KEYBOARD_INPUT.MAP_LIST;         -- keyboard bindings

   subtype LINE             is CURSOR.LINE;
   subtype COLUMN           is CURSOR.COLUMN;
   subtype COLUMN_INCREMENT is CURSOR.COLUMN_INCREMENT;   -- relative col. offset
   subtype LAYOUT_INDEX     is CURSOR.LAYOUT_INDEX;       -- layout array index

   -- Visual attributes of the cursor

   type INTENSITY_SETTING  is new CURSOR.INTENSITY_SETTING;
   type BLINK_SETTING      is new CURSOR.BLINK_SETTING;
   type SCREEN_COLORS      is new CURSOR.SCREEN_COLORS;

   type V_STRING is new CURSOR.V_STRING;                 -- variable length string


   --------------------------------------------------------------------------
   -- The following record types are from ANSI_CURSOR_SERVICES.   Merely
   -- deriving them here would cause the record's fields to be of types
   -- from ANSI_CURSOR_SERVICES rather than of the corresponding types
   -- derived above.  Because of this drawback of record derivation,
   -- these records are simply recreated here so that their fields
   -- are of the above derived types.  This package handles all the
   -- necessary type conversions.  The pending release of Ada 9X is expected
   -- to address this problem.
   --------------------------------------------------------------------------

   type POSITION is    -- Screen position of cursor
      record
        ROW : LINE   := 1;
        COL : COLUMN := 1;
      end record;
```

```
type TEXT_COLOR is      -- Visual attributes of the cursor
   record
      BACKGROUND : SCREEN_COLORS      := BLACK;
      FOREGROUND : SCREEN_COLORS      := WHITE;
      INTENSITY  : INTENSITY_SETTING := DIM;
      BLINKING   : BLINK_SETTING       := NO_BLINK;
   end record;

type ATTRIBUTE is
   record
      POS   : POSITION;
      COLOR : TEXT_COLOR;
   end record;

type TEXT_REC is     -- Formatted string (i.e., text/attribute pair)
   record
      TEXT : V_STRING;
      ATT  : ATTRIBUTE;
   end record;

type LAYOUT is array( LAYOUT_INDEX range <> ) of TEXT_REC;

type TERMINAL is private;


-----------------------------------------------------------------------
-- Exceptions
-----------------------------------------------------------------------

IO_ERROR : exception;    -- IO_ERROR is raised whenever anything goes wrong
                         -- within ANSI_TERMINAL_SERVICES or its supporting
                         -- packages or when the user requests an illegal
                         -- I/O service.


-----------------------------------------------------------------------
-- Functions to return private TERMINAL data
-----------------------------------------------------------------------

function THIS_TERM_PORT      ( THIS_TERM : in TERMINAL ) return PORT_DATA;
function THIS_TERM_MAP_LIST ( THIS_TERM : in TERMINAL ) return MAP_LIST;
function THIS_TERM_NUM_LINES( THIS_TERM : in TERMINAL ) return LINE;
function THIS_TERM_NUM_COLS ( THIS_TERM : in TERMINAL ) return COLUMN;


-----------------------------------------------------------------------
-- TO_V_STRING
-----------------------------------------------------------------------
-- Exceptions:
--    None.
-----------------------------------------------------------------------

function TO_V_STRING( S : in    STRING ) return V_STRING;


-----------------------------------------------------------------------
-- TO_STRING
-----------------------------------------------------------------------
-- Exceptions:
--    None.
-----------------------------------------------------------------------
```

```
function TO_STRING( V_STR : in      V_STRING ) return STRING;
```

```
--------------------------------------------------------------------
-- CREATE_TERMINAL
--------------------------------------------------------------------
-- Purpose:
--   This function returns a dynamically created terminal manager which
--   provides all the mechanisms needed for protected I/O in a multitasking
--   environment.  The function assigns a channel using the port name,
--   allocates and initializes keyboard and screen monitor tasks, allocates
--   shared data resources, maps the keyboard according to the specified
--   binding file, and initializes other terminal parameters.
-- Note:
--   After creating a terminal in this manner, users should explicitly set
--   the terminal display with a call to SET_CURSOR from the SCREEN package
--   below.
-- Exceptions:
--   IO_ERROR
--------------------------------------------------------------------

function CREATE_TERMINAL( PN        : in      PORT_NAME;
                          MAP_NAME  : in      STRING;
                          ROWS      : in      LINE      := MAX_LINE;
                          COLS      : in      COLUMN    := MAX_COLUMN )
                                      return TERMINAL;
```

```
--------------------------------------------------------------------
-- SHUTDOWN_TERMINAL
--------------------------------------------------------------------
-- Purpose:
--   This procedure deallocates a terminal's resources and aborts its
--   concurrency control mechanisms.
-- Exceptions:
--   IO_ERROR
--------------------------------------------------------------------

procedure SHUTDOWN_TERMINAL( T : in out TERMINAL );
```

```
--------------------------------------------------------------------
-- SCREEN
--------------------------------------------------------------------
-- Purpose:
--   This package serves as the interface through which users access
--   ANSI_TERMINAL_SERVICES' protected output services.
--
-- Effects:
--   - A call to any of these subroutines results in a rendezvous with the
--     screen monitor task.
--   - IO_ERROR is raised if any problems occur.
--
-- Performance:
--   - The LAYOUT version of PUT is optimized to display several formatted
--     strings with a single write.
--
--------------------------------------------------------------------
```

```
package SCREEN is


   -------------------------------------------------------------------
   -- CLEAR_SCREEN
   -------------------------------------------------------------------
   -- Purpose:
   --   This procedure clears the specified display and repositions the
   --   at (1,1).
   -- Exceptions:
   --   IO_ERROR
   -------------------------------------------------------------------

   procedure CLEAR_SCREEN( TERM : in      TERMINAL );




   -------------------------------------------------------------------
   -- CLEAR_TO_END_OF_LINE
   -------------------------------------------------------------------
   -- Exceptions:
   --   IO_ERROR
   -------------------------------------------------------------------

   procedure CLEAR_TO_END_OF_LINE( TERM : in      TERMINAL );




   -------------------------------------------------------------------
   -- SET_COLOR
   -------------------------------------------------------------------
   -- Exceptions:
   --   IO_ERROR
   -------------------------------------------------------------------

   procedure SET_COLOR( TERM  : in      TERMINAL;
                        COLOR : in      TEXT_COLOR );




   -------------------------------------------------------------------
   -- GET_COLOR
   -------------------------------------------------------------------
   -- Exceptions:
   --   IO_ERROR
   -------------------------------------------------------------------

   function GET_COLOR( TERM  : in      TERMINAL ) return TEXT_COLOR;




   -------------------------------------------------------------------
   -- SET_POSITION
   -------------------------------------------------------------------
   -- Exceptions:
   --   IO_ERROR
   -------------------------------------------------------------------

   procedure SET_POSITION( TERM : in      TERMINAL;
                           POS  : in      POSITION );
```

```
-------------------------------------------------------------------
-- SET_COLUMN
-------------------------------------------------------------------
-- Exceptions:
--    IO_ERROR
-------------------------------------------------------------------

procedure SET_COLUMN( TERM : in     TERMINAL;
                      INC  : in     COLUMN_INCREMENT );




-------------------------------------------------------------------
-- GET_POSITION
-------------------------------------------------------------------
-- Exceptions:
--    IO_ERROR
-------------------------------------------------------------------

function GET_POSITION( TERM : in     TERMINAL ) return POSITION;




-------------------------------------------------------------------
-- SET_CURSOR
-------------------------------------------------------------------
-- Exceptions:
--    IO_ERROR
-------------------------------------------------------------------

procedure SET_CURSOR( TERM : in     TERMINAL;
                      ATT  : in     ATTRIBUTE );




-------------------------------------------------------------------
-- GET_CURSOR
-------------------------------------------------------------------
-- Exceptions:
--    IO_ERROR
-------------------------------------------------------------------

function GET_CURSOR( TERM : in     TERMINAL ) return ATTRIBUTE;




-------------------------------------------------------------------
-- PUT_PORT
-------------------------------------------------------------------
-- Purpose:
--    This overloaded procedure writes a character or string to the given
--    terminal's assigned port.  Because the intended output target is a
--    port rather than a display, no cursor protection is provided.
-- Exceptions:
--    IO_ERROR
-------------------------------------------------------------------
```

```
procedure PUT_PORT( TERM : in     TERMINAL;
                    CHAR : in     CHARACTER );

procedure PUT_PORT( TERM : in     TERMINAL;
                    STR  : in     STRING );
```

```
--------------------------------------------------------------------
-- PUT_STR
--------------------------------------------------------------------
-- Purpose:
--   This procedure is used exclusively by the LINE_EDITOR supporting
--   package to echo user input and shift the cursor accordingly.  It
--   provides no cursor protection when called outside of LINE_EDITOR.
-- Exceptions:
--   IO_ERROR
--------------------------------------------------------------------
```

```
procedure PUT_STR( TERM : in     TERMINAL;
                   STR  : in     STRING;
                   INC  : in     CURSOR.PUT_STR_INCREMENT );
```

```
--------------------------------------------------------------------
-- PUT
--------------------------------------------------------------------
-- Purpose:
--   This overloaded procedure writes a character, formatted string, or
--   array of formatted strings to the display of the given terminal.
-- Exceptions:
--   IO_ERROR
--------------------------------------------------------------------
```

```
procedure PUT( TERM : in     TERMINAL;
               ATT  : in     ATTRIBUTE;
               CHAR : in     CHARACTER );

procedure PUT( TERM : in     TERMINAL;
               ATT  : in     ATTRIBUTE;
               STR  : in     STRING );

procedure PUT( TERM        : in     TERMINAL;
               LAYOUT_INFO : in     LAYOUT );
end SCREEN;
```

```
--------------------------------------------------------------------
-- KEYBD
--------------------------------------------------------------------
-- Purpose:
--   This package serves as the interface through which users access
--   ANSI_TERMINAL_SERVICES' protected input services.
--
-- Effects:
--   - Keyboard input routines operate upon the current cursor state.
--   - A call to any of these subroutines results in a rendezvous with the
--     keyboard monitor task.
```

```
--    - IO_ERROR is raised if any problems occur.
--
-- ------------------------------------------------------------------

package KEYBD is

   -- ----------------------------------------------------------------
   -- GET_LINE_EDIT
   -- ----------------------------------------------------------------
   -- Purpose:
   --   This procedure invokes the line editor service for the terminal and
   --   returns the edited string and the terminating keystroke.  The
   --   operations supported by the editor are listed in the specification of
   --   the LINE_EDITOR package.
   -- Exceptions:
   --   IO_ERROR
   -- ----------------------------------------------------------------

   procedure GET_LINE_EDIT( TERM        : in      TERMINAL;
                            STR         : in out STRING;
                            TERMINATOR  :     out ALL_KEYS);



   -- ----------------------------------------------------------------
   -- GET_NOECHO
   -- ----------------------------------------------------------------
   -- Purpose:
   --   This overloaded procedure reads a character or string from the given
   --   terminal's port.  This read does not echo the results or capture
   --   escape sequences.
   -- Exceptions:
   --   IO_ERROR
   -- ----------------------------------------------------------------

   procedure GET_NOECHO( TERM : in      TERMINAL;
                         CHAR :     out CHARACTER );

   procedure GET_NOECHO( TERM   : in      TERMINAL;
                         STR    :     out STRING;
                         LENGTH :     out INTEGER );



   -- ----------------------------------------------------------------
   -- GET_TIMED_NOECHO
   -- ----------------------------------------------------------------
   -- Purpose:
   --   This overloaded procedure reads a character or string from the given
   --   terminal's port.  This read does not echo the results or capture
   --   escape sequences.  If no response is detected in the given time
   --   interval, the read will terminate.
   -- Exceptions:
   --   IO_ERROR
   -- ----------------------------------------------------------------

   procedure GET_TIMED_NOECHO( TERM       : in      TERMINAL;
                               WAIT       : in      DURATION;
                               CHAR       :     out CHARACTER;
                               TIMED_OUT  :     out BOOLEAN );

   procedure GET_TIMED_NOECHO( TERM       : in      TERMINAL;
                               WAIT       : in      DURATION;
                               STR        :     out STRING;
```

A-9

```
                            LENGTH     :      out INTEGER;
                            TIMED_OUT :      out BOOLEAN );


        -------------------------------------------------------------------
        -- GET_KEY
        -------------------------------------------------------------------
        -- Purpose:
        --    This procedure reads a single keystroke from the terminal's keyboard,
        --    optionally echoes a printable response, and returns the corresponding
        --    logical key.
        -- Exceptions:
        --    IO_ERROR
        -------------------------------------------------------------------

        procedure GET_KEY( TERM       : in     TERMINAL;
                           KEY        :     out ALL_KEYS;
                           ECHO_FLAG : in     BOOLEAN );




        -------------------------------------------------------------------
        -- GET_TIMED_KEY
        -------------------------------------------------------------------
        -- Purpose:
        --    This procedure reads a single keystroke from the terminal's keyboard,
        --    optionally echoes a printable response, and returns the corresponding
        --    logical key.  If no response is detected in the given time interval,
        --    the read will terminate.
        -- Exceptions:
        --    IO_ERROR
        -------------------------------------------------------------------
        procedure GET_TIMED_KEY( TERM       : in     TERMINAL;
                                 WAIT       : in     DURATION;
                                 KEY        :     out ALL_KEYS;
                                 ECHO_FLAG : in     BOOLEAN;
                                 TIMED_OUT : in out BOOLEAN );
    end KEYBD;



        -------------------------------------------------------------------


private

    type KEYBD_CONTROL;       -- incomplete type declaration of keyboard task type
    type SCREEN_CONTROL;      -- incomplete type declaration of screen task type

    type KEYBD_ACCESS   is access KEYBD_CONTROL;  -- allows runtime task creation
    type SCREEN_ACCESS  is access SCREEN_CONTROL; -- allows runtime task creation
    type CURSOR_ACCESS  is access ATTRIBUTE;      -- shared cursor resource ptr
    type KEY_MAP_ACCESS is access MAP_LIST;       -- terminal's keyboard bindings


    type TERMINAL is   -- Terminal resource manager
        record
            PORT      : PORT_DATA;        -- host-independent port info
            KEY_MAP   : KEY_MAP_ACCESS;   -- keyboard bindings
            NUM_LINES : LINE;             -- lines the terminal supports
            NUM_COLS  : COLUMN;           -- columns the terminal supports
            KEYBD     : KEYBD_ACCESS;     -- controls access to keyboard
            SCREEN    : SCREEN_ACCESS;    -- controls access to screen
            CURSOR    : CURSOR_ACCESS;    -- cursor characteristics
```

```
        end record;


------------------------------------------------------------------------
-- To explicitly set priorities for the underlying KEYBD and SCREEN
-- task types, the user must remove the comment symbols below
-- AND before the two PRAGMA PRIORITY()s in the corresponding task specs.
------------------------------------------------------------------------

-- KEYBD_PRIORITY  : constant integer := 15;   -- range of values is
-- SCREEN_PRIORITY : constant integer := 14;   --   system dependent




------------------------------------------------------------------------
-- SCREEN_CONTROL
------------------------------------------------------------------------
-- This monitor task protects the cursor resource when performing
-- output.  Its entries correspond to output services users request via
-- procedures from the SCREEN package above.
------------------------------------------------------------------------

task type SCREEN_CONTROL is

-- pragma PRIORITY ( SCREEN_PRIORITY );

  -- Copy in terminal parameters
  entry COPY_TERMINAL_PARAMS( T : in      TERMINAL );

  -- Clear the terminal's screen & move cursor to position (1,1)
  entry CLEAR_SCREEN( ERROR :    out BOOLEAN );

  -- Clear in-line text right of the cursor
  entry CLEAR_TO_END_OF_LINE( ERROR :    out BOOLEAN );

  -- Set the terminal's visual attributes
  entry SET_COLOR( C      : in     TEXT_COLOR;
                   ERROR :    out BOOLEAN );

  -- Copy out the cursor's visual attributes
  entry GET_COLOR( C :    out TEXT_COLOR );

  -- Reposition the cursor
  entry SET_POSITION( POS   : in     POSITION;
                      ERROR :    out BOOLEAN );

  -- Set the cursor's column using a relative column offset
  entry SET_COLUMN( INCREMENT : in     COLUMN_INCREMENT;
                    ERROR     :    out BOOLEAN );

  -- Copy out the cursor's position
  entry GET_POSITION( POS   :    out POSITION );

  -- Set the cursor's positional and visual attributes
  entry SET_CURSOR( ATT   : in     ATTRIBUTE;
                    ERROR :    out BOOLEAN );

  -- Copy out the cursor's positional and visual attributes
  entry GET_CURSOR( ATT   :    out ATTRIBUTE );

  -- Write a stream of text to a port
  entry PUT_PORT( STR   : in     STRING;
                  ERROR :    out BOOLEAN );

  -- Write a string and (optionally) shift the cursor.
```

```
-- NO CURSOR PROTECTION IS PROVIDED.
entry PUT_STR( STR       : in      STRING;
               INCREMENT : in      CURSOR.PUT_STR_INCREMENT;
               ERROR     : in out BOOLEAN );


-- Safely write a string to the display using the given attributes
entry PUT( ATT   : in      ATTRIBUTE;
           STR   : in      STRING;
           ERROR : in out BOOLEAN );


-- Safely write an array of formatted text to the screen
entry PUT( LAYOUT_INFO : in      LAYOUT;
           ERROR       : in out BOOLEAN );


-- Abort this monitor task
entry SHUTDOWN;

end SCREEN_CONTROL;




--------------------------------------------------------------------------
-- KEYBD_CONTROL
--------------------------------------------------------------------------
-- This monitor task protects the cursor resource when performing
-- input.  Its entries correspond to input services users access via
-- procedures from the KEYBD package below.
--------------------------------------------------------------------------


task type KEYBD_CONTROL is

-- pragma PRIORITY( KEYBD_PRIORITY );

   -- Copy in terminal parameters
   entry COPY_TERMINAL_PARAMS( T : in      TERMINAL );

   -- Invoke the line editor service
   entry GET_LINE_EDIT( STR        : in out STRING;
                        TERMINATOR :     out ALL_KEYS;
                        ERROR      :     out BOOLEAN );

   -- Read a stream of input without echo or escape sequence detection
   entry GET_NOECHO( STR    :     out STRING;
                     LENGTH :     out INTEGER;
                     ERROR  :     out BOOLEAN );

   -- Timed stream read without echo or escape sequence detection
   entry GET_TIMED_NOECHO( WAIT      : in  DURATION;
                           STR       :     out STRING;
                           LENGTH    :     out INTEGER;
                           TIMED_OUT :     out BOOLEAN;
                           ERROR     :     out BOOLEAN );

   -- Read a single keystroke with escape sequence detection and
   --    optional screen echo
   entry GET_KEY( KEY       :     out ALL_KEYS;
                  ECHO_FLAG : in      BOOLEAN;
                  ERROR     : in out BOOLEAN );

   -- Timed read of a single keystroke with escape sequence detection and
   --    optional screen echo
   entry GET_TIMED_KEY ( WAIT      : in      DURATION;
                         KEY       :     out ALL_KEYS;
                         ECHO_FLAG : in      BOOLEAN;
                         TIMED_OUT : in out BOOLEAN;
                         ERROR     : in out BOOLEAN );
```

```
        -- Abort keyboard monitor task
        entry SHUTDOWN;

    end KEYBD_CONTROL;
end ANSI_TERMINAL_SERVICES;
```

APPENDIX B

FORMAT FOR KEYBOARD MAPPING FILE

The keyboard mapping file is read to initialize the MAP_LIST type. This type binds the mappable *logical keys* to the particular *escape sequences* the given keyboard generates. Alternatively, the type may be used to bind other unused keys to the functionality associated with a *logical key*. For example, if the keyboard does not have a **Home** key, the HOME *logical key* could be mapped to **F1**, **Control-H**, or any other suitable unused key. An example file is given in Appendix C. The following rules apply to the keyboard mapping file:

1. The file may have any number of comment lines denoted by the symbol "--" beginning in the first column.

2. Within a line, any text occurring after the mapping specification is treated as a comment.

3. No spaces are allowed in any mapping specification.

4. Function *logical keys* must be mapped sequentially beginning with FK1.

5. Edit *logical keys* may be mapped in any order.

6. Unmapped function *logical keys* will be initialized to a string of blank characters (ASCII 32).

7. Unmapped edit *logical keys* will be initialized as follows:

```
UP_ARROW     => /E[A
DOWN_ARROW   => /E[B
RIGHT_ARROW  => /E[C
LEFT_ARROW   => /E[D
TAB          => #9    (Control-I)
BACKSPACE    => #8    (Control-H)
DELETE       => #127  (DEL)
others       => #32   (blanks)
```

8. A function *logical key* (FK1 .. FK100) may only be bound to an *escape sequence*. The mapping is specified as FKn=/E{c} where

   • n is the number (1..100) of the function *logical key* being mapped

   • /E is required and denotes the ESC character (ASCII 27) introducing the sequence

   • {c} is a string of one or more nonspace printable ASCII characters

FORMAT FOR KEYBOARD MAPPING FILE

9. The 14 mappable edit *logical keys* may be bound to either *escape sequences*, control characters (ASCII 0 - ASCII 31), or the delete character (ASCII 127). The mapping is specified as either XX = #n or XX = /E{c} where

- XX is a two-letter abbreviation for the mappable edit *logical keys* in type ALL_KEYS:

IN = INSERT	HO = HOME	EN = ENDD
RA = RIGHT_ARROW	LA = LEFT_ARROW	UA = UP_ARROW
DA = DOWN_ARROW	PU = PAGE_UP	PD = PAGE_DOWN
TB = TAB	TR = TAB_REVERSE	BS = BACKSPACE
EL = ERASE_LINE	DL = DELETE	

- #n indicates that the *logical key* is being mapped to the ASCII character numbered n. This character may be a control character (ASCII 0 - ASCII 31) or the delete character (ASCII 127)

- /E indicates that the *logical key* is being mapped to an *escape sequence* where /E denotes the ESC character (ASCII 27) introducing the sequence

- {c} is a string of one or more nonspace printable ASCII characters

APPENDIX C

EXAMPLE KEYBOARD MAPPING FILE

```
--
-- VT320 Keyboard mappings
--
--
IN=/E[2~          -- map INSERT to <ESC>[2~
EN=#5             -- map ENDD to Control-E, ASCII 5
HO=#4             -- map HOME to Control-D, ASCII 4
RA=/E[C           -- map RIGHT_ARROW to <ESC>[C
LA=/E[D           -- map LEFT_ARROW
UA=/E[A           -- map UP_ARROW
DA=/E[B           -- map DOWN_ARROW
BS=#8             -- map BACKSPACE to Control-H, ASCII 8
DL=#127           -- map DELETE to DEL character, ASCII 127
TB=#9             -- map TAB to Control-I, ASCII 9
TR=#2             -- map TAB_REVERSE to Control-B, ASCII 2
EL=#22            -- map ERASE_LINE to Control-V, ASCII 22
PU=/E[5~          -- map PAGE_UP to "Prev Screen" key which generates <ESC>[5~
PD=/E[6~          -- map PAGE_DOWN to "Next Screen" key
FK1=/EOP          -- map FK1 to "PF1" key
FK2=/EOQ          -- PF2
FK3=/EOR          -- PF3
FK4=/EOS          -- PF4
FK5=/E[1~         -- map FK5 to "Find" key
FK6=/E[3~         -- REMOVE
FK7=/E[18~        -- F7
FK8=/E[19~
FK9=/E[20~
FK10=/E[21~
FK11=/E[23~
FK12=/E[24~
FK13=/E[25~
FK14=/E[26~
FK15=/E[28~
FK16=/E[29~
FK17=/E[31~
FK18=/E[32~
FK19=/E[33~
FK20=/E[34~       -- F20
FK21=/E[4~        -- map FK21 to "Select" key
```

APPENDIX D

FORMAT FOR CONSTANTS FILE

The ASCII file **ANSI.DAT** is read by the ANSI_CONSTANTS package at elaboration time in order to initialize several critical constants [2.8] needed by ANSI_TERMINAL_SERVICES and its supporting packages. The format of the file is quite simple; each line consists of an integer and optional commenting text. Because each line is associated with a single constant, anything appearing after the integer value is ignored. The name of the constant to which the value will be assigned is therefore generally listed after the integer. Since the constants will be initialized in sequence by ANSI_CONSTANTS, the values in the constants file must be in the order listed below. The default **ANSI.DAT** file is given in Appendix E.

1. MAX_PORT_NAME_SIZE
2. MAX_CHANNELS
3. MAX_ESC_LEN
4. MAX_LINE
5. MAX_COLUMN
6. MAX_LONG_STRING_SIZE
7. MAX_STRINGS_IN_LAYOUT

APPENDIX E

DEFAULT CONSTANTS FILE

E-2

```
15       -- MAX_PORT_NAME_SIZE
16       -- MAX_CHANNELS
5        -- MAX_ESC_LEN
25       -- MAX_LINE
80       -- MAX_COLUMN
1024     -- MAX_LONG_STRING_SIZE
30       -- MAX_STRINGS_IN_LAYOUT
```

APPENDIX F

EXAMPLE APPLICATION CODE

```
-------------------------------------------------------------------------
--
--                Dahlgren Division, Naval Surface Warfare Center
--
--
-- Author:                   C. Chalkley
--
-- Department:               Combat System Technologies Branch (N35)
--
-- Revision History:
--    1/31/92               C. Chalkley
--       - EXAMPLE driver procedure completed
--
-------------------------------------------------------------------------




-------------------------------------------------------------------------
-- EXAMPLE
-------------------------------------------------------------------------
-- Purpose:
--    This procedure is an example application of ANSI_TERMINAL_SERVICES.  Its
--    embedded tasks illustrate the proper use of many ANSI_TERMINAL_SERVICES'
--    I/O routines
--
-- Effects:
--    - The keyboard mapping file "vt320.dat" must exist.
--    - This version executes on a DEC MicroVAX and opens a channel using the
--      SS$COMMAND port name.
--    - This procedure includes three tasks which will concurrently perform
--      I/O on a common terminal screen.
--
-- Performance:
--    - Results my vary depending upon assigned task priorities and system loads.
-------------------------------------------------------------------------


with SYSTEM;
with TEXT_IO;
with CALENDAR;
with ANSI_TERMINAL_SERVICES;

procedure EXAMPLE is
   pragma priority( SYSTEM.PRIORITY'first );    -- proc body gets lowest priority

   package ANSI renames ANSI_TERMINAL_SERVICES;

   task TIMER is
      entry START;        -- Start counting off seconds
      entry STOP;         -- Stop counting
   end TIMER;

   task READER is
      entry START;                               -- Start line editor
      entry SET_DANGER( NEW_SETTING : in BOOLEAN );   -- Sets the danger flag
   end READER;

   task WARNING is
      entry START;                               -- Start delayed warning
      pragma priority( SYSTEM.PRIORITY'last );   -- give WARNING top system priority
   end WARNING;

   PORT_NAME  : ANSI.PORT_NAME  := "SYS$COMMAND     ";  -- assigned port (VMS)
   MAP_NAME   : constant STRING := "vt320.dat";         -- keyboard binding file
   THIS_TERM  : ANSI.TERMINAL :=
               ANSI.CREATE_TERMINAL( PORT_NAME, MAP_NAME,
```

```
                                    ANSI.MAX_LINE, ANSI.MAX_COLUMN );


---------------------------------------------------------------------
-- TIMER
---------------------------------------------------------------------
-- Purpose:
--   This task periodically updates a clock counter in the upper left
--   corner of the terminal display.
--
-- Exceptions:
--   None.
---------------------------------------------------------------------


task body TIMER is
    CUR_TIME   : CALENDAR.TIME;          -- system time variable
    SEC_COUNT  : CALENDAR.DAY_DURATION;  -- seconds since midnight
    HOURS      : INTEGER := 0;           -- clock counters
    MINUTES    : INTEGER := 0;
    SECONDS    : INTEGER := 0;
begin

    accept START;      -- wait here until told to start

    ANSI.SCREEN.PUT(THIS_TERM,
        ((1,1), (ANSI.BLACK, ANSI.WHITE, ANSI.BOLD, ANSI.NO_BLINK)), "TIME:");

    loop               -- edit until a threat arrives

      select
         accept STOP;
         exit;         -- exit loop & terminate the task

      or

         delay 1.0;    -- wait one second for STOP request

         CUR_TIME   := CALENDAR.CLOCK;                      -- get current time
         SEC_COUNT  := CALENDAR.SECONDS( CUR_TIME );        -- convert time
         HOURS      := INTEGER(SEC_COUNT) / 3600;
         MINUTES    := INTEGER(SEC_COUNT) rem 3600 / 60;
         SECONDS    := INTEGER(SEC_COUNT) rem 60;

         -- display updated time
         ANSI.SCREEN.PUT(THIS_TERM,
             ((1,8), (ANSI.BLACK, ANSI.WHITE, ANSI.DIM, ANSI.NO_BLINK)),
             INTEGER'image( HOURS ) & ":" &
             INTEGER'image(MINUTES) & ":" &
             INTEGER'image(SECONDS) & "  ");
      end select;
    end loop;
end TIMER;


---------------------------------------------------------------------
-- READER
---------------------------------------------------------------------
-- Purpose:
--   This task repeatedly invokes the line editor until the user elects to
--   stop editing in the presence of a threat.  It relies upon the WARNING
--   task to inform it of the presence of a threat.
--
-- Exceptions:
--   None.
---------------------------------------------------------------------
```

```
task body READER is
    STR         : STRING(1..12);                     -- string to edit
    TERMINATOR  : ANSI.ALL_KEYS := ANSI.NUL;         -- editor terminator keystroke
    RESPONSE    : ANSI.ALL_KEYS := ANSI.NUL;         -- threat response keystroke
    TIME_OUT    : BOOLEAN;                           -- used for timed read
    DANGER      : BOOLEAN := false;                  -- presence of a threat

begin

    accept START;       -- Synchronize before editing begins

    -- display line editor prompt
    ANSI.SCREEN.PUT(THIS_TERM,
        ((13,18), (ANSI.BLACK, ANSI.WHITE, ANSI.BOLD, ANSI.NO_BLINK)),
        "EDIT SOME TEXT:");

    loop

        STR(1..12) := "Default text";    -- string to be edited

        select
          accept SET_DANGER( NEW_SETTING : in BOOLEAN ) do
            DANGER := NEW_SETTING;
          end SET_DANGER;

        else

          if DANGER then
            RESPONSE := TERMINATOR;

            -- Erase prompt as user responds to warning

            ANSI.SCREEN.SET_POSITION( THIS_TERM, (13,18));
            ANSI.SCREEN.CLEAR_TO_END_OF_LINE( THIS_TERM );

            -- Force user to respond to warning message if line editor
            -- terminator didn't address the threat

            while (RESPONSE not in ANSI.FK1 .. ANSI.FK3) loop
              ANSI.SCREEN.PUT(THIS_TERM,
                  ((10,27), (ANSI.BLACK, ANSI.RED, ANSI.BOLD, ANSI.BLINK)),
                  ASCII.BEL & "*** RESPOND TO THREAT ***");
              ANSI.SCREEN.SET_POSITION( THIS_TERM, (10,27));

              -- Get a keystroke - if no response in 5 seconds, beep & try again
              ANSI.KEYBD.GET_TIMED_KEY(THIS_TERM, 5.0, RESPONSE, false, TIME_OUT);

            end loop;

            -- take appropriate action on user's response to the warning

            case RESPONSE is
              when ANSI.FK1 =>
                  TIMER.STOP;
                  ANSI.SCREEN.CLEAR_SCREEN( THIS_TERM );
                  ANSI.SCREEN.PUT(THIS_TERM,
                      ((5,1), (ANSI.BLACK, ANSI.WHITE, ANSI.DIM, ANSI.NO_BLINK)),
                      "Threat Assessment .....");
                  exit;

              when ANSI.FK2 =>
                  TIMER.STOP;
                  ANSI.SCREEN.CLEAR_SCREEN( THIS_TERM );
                  ANSI.SCREEN.PUT(THIS_TERM,
                      ((5,1), (ANSI.BLACK, ANSI.WHITE, ANSI.DIM, ANSI.NO_BLINK)),
                      "Initiating Defense .....");
```

```
                    exit;

              when ANSI.FK3 =>
                   ANSI.SCREEN.SET_POSITION( THIS_TERM, (10,27));
                   ANSI.SCREEN.CLEAR_TO_END_OF_LINE( THIS_TERM );
                   ANSI.SCREEN.PUT(THIS_TERM,
                        ((13,18), (ANSI.BLACK, ANSI.WHITE, ANSI.BOLD, ANSI.NO_BLINK)),
                        "EDIT SOME TEXT:");
                   ANSI.SCREEN.SET_POSITION( THIS_TERM, (13,35));
                   ANSI.KEYBD.GET_LINE_EDIT( THIS_TERM, STR, TERMINATOR );

              when others   => null;
            end case;

         else

            -- set cursor position & call line editing service
            ANSI.SCREEN.SET_POSITION(THIS_TERM, (13,35));
            ANSI.KEYBD.GET_LINE_EDIT( THIS_TERM, STR, TERMINATOR );
         end if;
       end select;
    end loop;
 end READER;




 ---------------------------------------------------------------------------
 -- WARNING
 ---------------------------------------------------------------------------
 -- Purpose:
 --   This task displays a warning message after a short delay and
 --   communicates the presence of a threat to the READER task.
 --
 -- Exceptions:
 --   None.
 ---------------------------------------------------------------------------

 task body WARNING is

    WARNING_MESSAGES : ANSI.LAYOUT(1..4);

 begin

    accept START;      -- wait until told to start

    WARNING_MESSAGES :=
       ((ANSI.TO_V_STRING( ASCII.BEL & "*** WARNING  :  NEW LAUNCH DETECTED ***" ),
            ((5,20), (ANSI.BLACK, ANSI.RED, ANSI.BOLD, ANSI.BLINK))),
        (ANSI.TO_V_STRING( "PRESS: F1 for threat assessment" ),
            ((6,24), (ANSI.BLACK, ANSI.YELLOW, ANSI.BOLD, ANSI.NO_BLINK))),
        (ANSI.TO_V_STRING( "       F2 to defend" ),
            ((7,24), (ANSI.BLACK, ANSI.YELLOW, ANSI.BOLD, ANSI.NO_BLINK))),
        (ANSI.TO_V_STRING( "       F3 to edit again" ),
            ((8,24), (ANSI.BLACK, ANSI.YELLOW, ANSI.BOLD, ANSI.NO_BLINK))));

    delay 5.0;        -- do nothing & then display urgent message

    ANSI.SCREEN.PUT( THIS_TERM, WARNING_MESSAGES );

    READER.SET_DANGER( true );        -- inform READER task of threat presence

 end WARNING;


 begin   -- EXAMPLE
```

```
    ANSI.SCREEN.CLEAR_SCREEN( THIS_TERM );
    ANSI.SCREEN.SET_CURSOR( THIS_TERM,
        ((1,1), (ANSI.BLACK, ANSI.WHITE, ANSI.DIM, ANSI.NO_BLINK)));

    TIMER.START;                    -- Start all tasks
    READER.START;
    WARNING.START;

    while READER'callable loop   -- Do nothing until tasks are finished
        delay 1.0;
    end loop;

    ANSI.SCREEN.SET_POSITION( THIS_TERM, (8,1));
    ANSI.SHUTDOWN_TERMINAL( THIS_TERM );    -- deallocate terminal attributes &
                                            --   terminate I/O monitor tasks

exception
    when ANSI.IO_ERROR =>
        TEXT_IO.PUT_LINE("Illegal I/O request");
    when others =>
        TEXT_IO.PUT_LINE("Error within procedure EXAMPLE");
end EXAMPLE;
```

DISTRIBUTION

	Copies		Copies
ATTN OP-094H	1	DEFENSE TECHNICAL	
CHIEF OF NAVAL OPERATIONS		INFORMATION CENTER	
DEPARTMENT OF THE NAVY		CAMERON STATION	
WASHINGTON DC 20350-5000		ALEXANDRIA VA 22304-6104	12
ATTN SPAWAR-30T	1	ADA INFORMATION	
SPAWAR-2312	1	CLEARINGHOUSE	
COMMANDER		C/O IIT RESEARCH INSTITUTE	
SPACE AND NAVAL WARFARE		4600 FORBES BOULEVARD	
SYSTEMS COMMAND		LANHAM MD 20706-4320	1
WASHINGTON DC 20363-5100			
		ADA JOINT PROGRAM OFFICE	
ATTN SEA-06	1	ROOM 3E114	
SEA-06D	1	THE PENTAGON	
SEA-06K	1	WASHINGTON DC 20301-3081	1
SEA-06KR	1		
COMMANDER		ATTN GIFT AND EXCHANGE	
NAVAL SEA SYSTEMS COMMAND		DIVISION	4
NAVAL SEA SYSTEMS COMMAND		LIBRARY OF CONGRESS	
HEADQUARTERS		WASHINGTON DC 20540	
2531 NATIONAL CITY BLDG 3			
WASHINGTON DC 20362-5160		ATTN LEW ZITZMAN	1
		APPLIED PHYSICS LABORATORY	
ATTN PMS-400B	1	THE JOHNS HOPKINS UNIVERSITY	
PMS-400B3	1	LAUREL MD 20723-6099	
PMS-400B5	1		
PMS-412	1	ATTN STAN RALPH	1
DEPARTMENT OF THE NAVY		GOVERNMENT ELECTRONIC	
AEGIS PROGRAM OFFICE		SYSTEMS DIVISION	
2531 NATIONAL CENTER BLDG 3		GENERAL ELECTRIC COMPANY	
WASHINGTON DC 20362-5160		MOORESTOWN NJ 08057	
CENTER FOR NAVAL ANALYSES		ATTN JOE CARUSO	2
4401 FORD AVE		COMPUTER SCIENCES	
ALEXANDRIA VA 22302-0268	1	CORPORATION	
		4001 OAK MANOR OFFICE PARK	
		SUITE 201	
		KING GEORGE VA 22485	

DISTRIBUTION (Continued)

	Copies		Copies
ATTN RALPH MATTEI		INTERNAL DISTRIBUTION	
(MAIL CODE 43)	2	(CONTINUED)	
COMPUTER SCIENCES		N42	1
CORPORATION		U30	1
203 WEST ROUTE 38		U303	1
MOORESTOWN NJ 08057		U33	1
INTERNAL DISTRIBUTION			
D	1		
D4	1		
E06	1		
E211 GREEN	1		
E231	3		
E232	2		
E261 WAITS	1		
E32 GIDEP	1		
F	1		
F30	1		
G	1		
G70	1		
J	1		
J10	1		
J12	5		
J12 CHALKLEY	3		
J14	1		
K	1		
K10	1		
N	1		
N02	1		
N04	1		
N05	1		
N06	1		
N10	1		
N20	1		
N22	1		
N24	1		
N30	1		
N304	1		
N305	1		
N33	1		
N35	25		
N40	1		

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE February 1992	3. REPORT TYPE AND DATES COVERED

4. TITLE AND SUBTITLE Ada-Based Multitasking Terminal I/O	5. FUNDING NUMBERS
6. AUTHOR(S) John C. Chalkley Michael W. Masters	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Surface Warfare Center Dahlgren Division (Code J12) Dahlgren, VA 22448-5000	8. PERFORMING ORGANIZATION REPORT NUMBER NAVSWC TR 91-783

9. SPONSORING/MONITORING AGENCY NAME(S) AND	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This report describes "ANSI Terminal Services," a reusable Ada-based layered approach to providing protected asynchronous terminal I/O in a multitasking environment. The traditional single-threaded, sequential programming model is inadequate for Ada multitask programs, particularly for real-time applications. An application is likely to contain several tasks needing to perform output concurrently. Any shared resources contained in the I/O software must, therefore, be protected through some form of mutual exclusion. Also, an application will likely need to support input operations concurrently with outputs. This requirement is especially important since Ada implementations on most current operating systems are process synchronous, rather than task synchronous/process asynchronous. As a result, they cause the entire user process to block on an input request. Thus, other program tasks that may be performing critical functions are suspended until user input is complete. Such a result is clearly inappropriate in a real-time environment. ANSI Terminal Services solves these problems and provides both a standard programmer interface and the necessary multitasking concurrency control.

14. SUBJECT TERMS			15. NUMBER OF PAGES 121
ANSI Terminal Services I/O concurrency	Ada synchronous terminal	multitasking asynchronous	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102