```
UU      UU      SSSSSSSS    EEEEEEEEEE    RRRRRRRR
UU      UU      SSSSSSSS    EEEEEEEEEE    RRRRRRRR
UU      UU    SS            EE            RR      RR
UU      UU    SS            EE            RR      RR
UU      UU    SS            EE            RR      RR
UU      UU    SS            EE            RR      RR
UU      UU      SSSSSS      EEEEEEEE      RRRRRRRR
UU      UU      SSSSSS      EEEEEEEE      RRRRRRRR
UU      UU            SS    EE            RR  RR
UU      UU            SS    EE            RR  RR
UU      UU            SS    EE            RR    RR
UU      UU            SS    EE            RR    RR
UUUUUUUUUU    SSSSSSSS      EEEEEEEEEE    RR      RR
UUUUUUUUUU    SSSSSSSS      EEEEEEEEEE    RR      RR



LL            PPPPPPPP      TTTTTTTTTT                  222222
LL            PPPPPPPP      TTTTTTTTTT                  222222
LL            PP      PP        TT                    22      22
LL            PP      PP        TT                    22      22
LL            PP      PP        TT                            22
LL            PP      PP        TT                            22
LL            PPPPPPPP          TT                          22
LL            PPPPPPPP          TT                          22
LL            PP                TT                        22
LL            PP                TT                        22
LL            PP                TT              ....    22
LL            PP                TT              ....    22
LLLLLLLLLL    PP                TT              ....    2222222222
LLLLLLLLLL    PP                TT              ....    2222222222



*START* Job USER Req #296 for EGB    Date 15-Feb-84 12:17:27 Monitor: //, TOPS-2
File RM:<JIM.ENV>USER.LPT.2, created: 29-Jan-84 21:21:43
        printed: 15-Feb-84 12:17:28
Job parameters: Request created:15-Feb-84 11:14:50   Page limit:45   Forms:NORMAL   A
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:ASCI
```

# Chapter 1
# User Interface

User interaction with the system and his own programs is through the editor. As a result, understanding the basic concepts of the editor is necessary to the use of the remainder of the environment. The users' investment in learning these facilities is repaid in increased functionality and more uniform interface.

The user is primarily interested in manipulating the entities that make up the environment. The user interface is concerned with providing an orderly and convenient method of expressing these manipulations. The user communicates by typing characters (or function keys or moving a mouse). Though system entities are often presentable in a readable form, the objects themselves are not made up of the characters used to present them. As a result, the user interface is constructed to interact with the user through character editor and with the entities themselves in terms of their own representation. To accomplish this, the editor is separated into two layers:

1. The visible interface is a multi-window editor that provides a core set of facilities for handling user input, editing and screen management. This is called the Core Editor.

2. The type-specific, object-knowledgeable portion of the editor is called the Object Editor. Which object editor is used depends on the type of the object (entity). Although specific object types may require specific operations, there is a common set of operations requiring type knowledge that is provided by all (or most) object editors. These are referred to as object operations.

## 1.1. Core Editor Concepts

The Core Editor provides character editing facilities. This section is an attempt to define and briefly describe these.

### 1.1.1. Screen Structure

A Screen is the entire contents of the display at a particular time. Screens are made up of opaque rectangular areas, called Windows, arranged in a possibly overlapping pattern. More than one screen can be maintained by a session to facilitate changing from one multi-window activity to another (though not initially).

Windows are composed of character positions and, optionally, borders. Borders are used to visually delineate windows. The character positions represent a bounded rectangular region of the quarter-plane of an Image.

An image is an array (Natural) of lines, each consisting of an array (Natural) or characters. At any time, each image has a specific number of lines, each of which consists of a specific number of characters. Lines beyond the end of the image and characters beyond the end of lines are treated as blanks on the window. A _word_ is a portion of a line delimited by separator characters. Word boundaries are completely syntactic and are handled by the Core Editor, though word boundaries may correspond to finest-grain object boundaries.

An image is the user-readable representation of an entity in the system. One of the functions of the Core Editor-Object Editor combination is to provide mechanisms to reflect changes from the readable to the internal representation and back within the editing paradigm. The image is the Core Editor representation of the object. The underlying entity or object is always referred to as an object (independent of the distinctions in ENTITIES).

_Superwindows_ are collections of windows that are logically linked and maintained to be physically contiguous. Because of this logical connection, superwindows are commonly referred to as windows composed of windows. The most common example of a superwindow configuration has the following characteristics:

1. A window containing the image of an object to be edited. This is called an _object window_. It normally has top and side borders.
2. A _banner window_ that explains the purpose and status of the object windows. Normally presented in a different font than its associated object window, with side borders.
3. A _command window_ that is used for entering Ada statements to be compiled and executed to perform actions on the user's behalf. Normally has bottom and side borders.
4. The appearance of the whole is of a single box, surrounded by borders, with the command window separated from the object window by the banner.
5. The command window is an object window in its own right.

There is a system-managed output window that serves as the destination for general error messages and system output. Its associated banner is used to depict the state of the session.

**1.1.2. Cursors**

The physical screen has an apparent _cursor_, marking the current position of the user's focus of attention (from the editor's point of view). This cursor represents the following:

1. Screen position. The line and column of the cursor on the screen.
2. Window position. If the cursor is within the bounds of a defined window, it represents a line and column position in the window.
3. Image position. If the cursor is within a window, it represents a line and column position within the image associated with the window.

The editor defines operations that change the screen, window and image position of the cursor. Each window has a current window and image position, corresponding to the last cursor position on that window and image. Operations that change the current window directly (i.e. without moving the cursor) place the cursor at these saved window and image positions. Moving the cursor on an image or window causes the window to be scrolled to assure that the cursor is visible at its current image position. Moving the cursor on the screen allows motion that ignores the boundaries of the window. Moving the cursor on the screen doesn't change the window or image positions that are saved with a window until an operation depending on the cursor is performed, even though the cursor seems to have passed over the window. This is really quite intuitive when written well ...

A _mark_ is a saved image position corresponding to the position of the cursor on the image when it was saved. Marks are stored in terms of absolute image positions and do not change to adjust for inserted/deleted lines/characters.

### 1.1.3. Fonts and Designations

Each character that appears in a window is displayed in some _font_. The appearance characteristics of fonts vary from terminal device to terminal device, but different fonts on the same device commonly differ in boldness, brightness, video presentation (reverse or normal), underlining and blinking. More advanced devices allow traditional font distinctions such as italics. Specific choices are terminal-specific, but banners are typically represented in reverse-video, keywords are underlined or emboldened.

On each window, the characters that represent themselves presented in the font that is the default for the window. Other fonts are used to represent usages that are considered important by either the Core Editor or the specific Object Editor. The Ada Object Editor presents Ada keywords in a different font from the remainder of the program.

Regardless of object type, fonts are used to represent the following notions:

1. Non-printing characters. Each non-printing ASCII character can be represented by its traditional position in the Control-sequence. Each of these is printed as a font-changed version of its base character. For example, ASCII.SOH (aka Control-A) might be represented as a reverse-video A.

2. Elisions. In order to suppress detail and improve screen usage, it is possible to elide the presentation of an object. The elision is commonly represented by the ellipsis symbol ("..."), but can be represented by a phrase that provides more information. In either case, the characters used for the elision are presented in a distinguished font. All of the characters in the elision mark represent the entire elision, so

it is not meaningful to edit the elision mark with the intent of changing the underlying object. The Core Editor supports this notion by limiting the edit operations allowed on elisions.

3. Prompts. Prompts are used to represent places that the user may want or need to insert content. Traditional prompts consist of one or more characters at the beginning of the line to indicate the readiness of the system (or one of its programs) to accept input. The editor's use of prompts is an expansion of this notion to two dimensions and a more general editing paradigm. Rather than be limited to a specific leading character, prompts are strings print in place whose text suggests what might be typed to replace them. Because the contents of the prompt is typically a reminder, it disappears when any change is made to it, allowing the user to simply type the replacement without explicitly deleting the contents of the prompt.

4. Selections. Selection is a basic notion of the editor whereby the user can designate part of an object as the implicit operand by selecting it. Selections are displayed in a font that makes the bounds of the selection clear. It is possible to select the region of an image between two points using Core Editor operations or a logical sub-object using Object Editor operations. An object selection can always be used as a text selection, but the arbitrary boundaries of a text selection can make it inappropriate for object operations.

5. Errors. An error is a section of text marked by the object editor to indicate a problem of some sort. Except for their appearance, errors are edited as if they were normal text.

Elisions and prompts are treated specially and are exclusive of each other; other font uses are for user information and can appear in concert. The term _designation_ is used to indicate a section of text marked as an elision, prompt, or Error.

**1.1.4. Mechanisms**

The following mechanisms are provided to support editing operations that are not primarily dependent on the apparent objects on the screen.

A _keymap_ is a mechanism for binding a key or key-sequence to a specific action. Every key that the user hits is bound by this mechanism to some command. For example, the most common commands are character insertions that are mapped to the key labelled with the character, but by changing the keymap, it would be possible to implement a Dvorak keyboard without modifying the terminal.

A _macro_ is a sequence of saved editor commands that can be invoked together. Macros are appropriate for recording a set of actions for re-use later. It is expected that complicated operations, including those requiring parameter passing, will be done with Ada programs. Facilities are provided for saving macros with a session and for binding them to keys in the same manner as built-in commands are bound to keys.

A _Yank Buffer_ is a piece of an image that has been saved for later use, typically by a deletion operation.

An _Extended Command_ is any Ada fragment that is compiled and executed outside of the Core Editor. The Core Editor has no information about what each of these commands does, but saves the image corresponding to each in case the user wishes to repeat the same or similar operation.

A _Stack_ is a structure for saving a set of objects based on usage patterns. The primitive operations are:

1. Push. Add/move an item at the top of the stack.
2. Next. Examine the next item down the stack.
3. Previous. Examine the previous item up the stack.
4. Top. Examine the item at the top of the stack.

Next (previous) "wraps" to the top (bottom) when applied to the bottom (top). Stacks are used to store marks, windows, images, selections, yank buffer, and extended commands.

## 1.2. Object Editor

The object editor provides the transformations between the object and its image and vice versa. This is done by incremental parsing and pretty-printing operations. The object editor provides operations to commit the changes that have been made, reflecting them in the permanent version of the object. The changes that have been made can also be abandoned, reverting to the form of the object at the last commit.

The object editor provides tree-structured selection operations that understand the structure of the object being edited. These operations provide the ability to select objects, their parents (the containing object), next and previous brothers, and children. These selected objects serve as operands to move, copy, delete, elide and expand operations, as well as to type-specific tools outside of the editor.

### 1.2.1. Pointing

## 1.3. Ada Editor

While it possible to conceive of object editors for many types, the first and most important is the one for Ada. Because of its interaction with system structure and semantics, the Ada object editor provides operations and, in some cases, imposes restrictions that have no parallel in other objects.

#### 1.3.1. Commands

Commands are Ada objects with some additional requirements.

What is context, how is it used and set.

### 1.4. Program Execution

Jobs and Sessions.

### 1.5. Keys and Command Factoring

The command set of the editor that is bound to keys has been factored into a sets of operations and sets of objects. Each group of operations can be applied to a group of types by using the key that specifies the object type followed by the key for the operation. Default object types have been chosen to reduce the frequency of two-key sequences, and since the factoring doesn't occupy all possible keys (especially for terminals with function keys, etc.), it is possible to place commonly used sequences on single keys.

##### 1.5.0.1. Types

The set of object types is:
1. Character cursor. Character insertion, image position and marks.
2. Command. Command window and history.
3. Designation. Elisions, error and prompts.
4. Macro.
5. Line.
6. Screen cursor. Motion on the screen.
7. Selection. Both object and text selection.
8. Window.
9. Yank buffer.

##### 1.5.0.2. Operations

The following is a brief description of each of the classes of operations and the types that each applies to.

1. Planar movement (up, down, left, right)
    1. Cursor. Move user cursor on the image
    2. Screen cursor. Move user cursor on screen.
    3. Selection. Select parent, child or brothers.
    4. Window. Scroll the window over the image.

2. Relative positioning (next, previous, beginning_of, end_of)

      1. Designation.
      2. Line.
      3. Word.

3. Modification. (copy, delete, insert, move, transpose; capitalize, lower-case, upper-case)
   1. Character.
   2. Line.
   3. Selection.
   4. Word.

4. Stack. (next, previous, push, top)
   1. Command. Manipulate history. Push is implied by execution.
   2. Mark.
   3. Selection.
   4. Yank buffer.

More detail, including an initial key assignment for QWERTY-only keyboards is available in [BLS.CE.DOC]R1000_Commands.MSS.

### 1.6. Package Directory Operations

Directory packages serve a number of functions. Initial support for directory viewing and manipulation centers on support for semantic-preserving meta-operations that allow the user to build up Ada structures and traverse them. These operations are described in below. Following that, there is a discussion of more traditional directory operations, how they fit in, and their eventual transition to full editor functionality.

#### 1.6.1. Insertion Points, Installation and Elaboration

There is a difference between source, installed and elaborated. You can edit source, you can create places to put source, called insertion points, so that it can be installed. You can elaborate installed things. There is help for determining which of these states an object is in.

#### 1.6.2. Directory View and Attributes

Traditional directory services provide access to a variety of information to help remind the user of what is contained in the directory, when it was created or changed, how large it is, etc. The principal support for this is a procedure, List_Directory, that will print a list of objects in a directory accompanied by the appropriate attributes. The list will appear on the screen as an output window. In the absence of protection information, the name of the object is about the only thing that can be changed. This is done by explicit command. Support for a restricted directory object editor would only require the ability to delete objects, and change their names, protection and resource limits. Creating objects or changing their Ada characteristics (type of program unit or Ada type, parameters, etc.) would still require using the Ada Object editor.

## Table of Contents