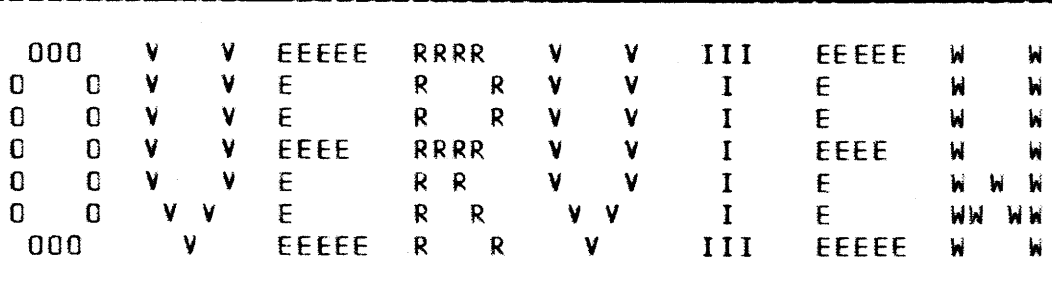
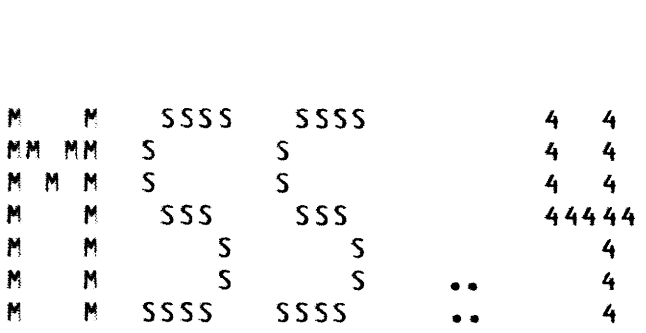
```
 III    N    N  TTTTT  EEEEE  RRRR   FFFFF    AAA    CCCC  EEEEE
  I     N    N    T    E      R   R  F       A   A  C      E
  I     NN   N    T    E      R   R  F       A   A  C      E
  I     N N  N    T    EEEE   RRRR   FFFF    A   A  C      EEEE
  I     N  NN    T    E      R R    F       AAAAA  C      E
  I     N    N    T    E      R  R   F       A   A  C      E
 III    N    N    T    EEEEE  R   R  F       A   A   CCCC  EEEEE



M    M   SSSS   SSSS          333
MM  MM   S      S             3   3
M M M    S      S                 3
M    M    SSS    SSS             3
M    M       S      S             3
M    M       S      S     ..    3   3
M    M   SSSS   SSSS      ..     333
```

START Job INTERF Req #297 for EGB Date 15-Feb-84 12:18:08 Monitor: //, TOPS
File RM:<JIM.ENV>INTERFACE.MSS.3, created: 30-Nov-83 21:53:10
        printed: 15-Feb-84 12:18:08
Job parameters: Request created:15-Feb-84 11:15:58 Page limit:63 Forms:NORMAL
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:ASC

```
@Section(Editor-Based User Interface)
@Begin(Itemize)
Editor based.

Object oriented.

Type knowledge.

Simple, yet complex.

Mechanisms
@Begin(SItemize)
Windows.

Completion.

Prompts.

Elision.
@End(SItemize)

Objects.
@Begin(SItemize)
Characters.

Words.

Objects.

Selections.

Cursors.
@End(SItemize)

@End(Itemize)
```





START Job INTERF Req #297 for EGB Date 15-Feb-84 12:18:08 Monitor: //, TOPS
File RM:<JIM.ENV>OVERVIEW.MSS.4, created: 30-Nov-83 20:32:12
printed: 15-Feb-84 12:18:11
Job parameters: Request created:15-Feb-84 11:15:58 Page limit:63 Forms:NORMAL
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:ASC

```
@Make(Article)
@Modify (Verbatim, above 1, below 1)
@Modify (Senumerate, above 0)
@Modify (Description, LeftMargin +8, Indent -8)
@String(DocumentTitle=<"..." Overview          >)
@String(Draft=<DRAFT 1>)
@Set(Page=1)
@Include(ada.mss)
@Include(process.mss)
@Include(interface.mss)
@Include(types.mss)
```

```
PPPP    RRRR     OOO     CCCC  EEEEE   SSSS   SSSS
P   P   R   R   O   O   C      E      S      S
P   P   R   R   O   O   C      E      S      S
PPPP    RRRR    O   O   C      EEEE    SSS    SSS
P       R R     O   O   C      E          S      S
P       R  R    O   O   C      E          S      S
P       R   R    OOO     CCCC  EEEEE  SSSS   SSSS



M   M   SSSS   SSSS          1    55555
MM MM   S      S            11    5
M M M   S      S             1    555
M   M    SSS    SSS          1       5
M   M       S      S         1        5
M   M       S      S    ..   1    5   5
M   M   SSSS   SSSS     ..   111   555



*START* Job INTERF Req #297 for EGB    Date 15-Feb-84 12:18:08 Monitor: //, TOPS
File RM:<JIM.ENV>PROCESS.MSS.15, created:  9-Jan-84 18:34:12
        printed: 15-Feb-84 12:18:16
Job parameters: Request created:15-Feb-84 11:15:58   Page limit:63   Forms:NORMAL
File parameters: Copy: 1 of 1   Spacing:SINGLE   File format:ASCII   Print mode:ASC
```

@Section(Users, Groups, Sessions, and Jobs)
Users, Groups, Sessions and Jobs are very simple representations for notions of interest to users of the environment. They all serve to identify tasks and objects on the basis of who created them.

@SubSection(Users)
A user is an object that represents the human user in the system. Its primary purpose is to provide a domain for system access authentication and object access control.

The environment associates information with the user that makes it possible for him to tailor the user interface and resume sessions in a desired state. Specifically, each user is associated with:
@Begin(SEnumerate)
The set of objects and packages that he has created. This includes sessions, files, programs, etc. The user is said to "own" such objects and packages.

A home package in the package directory system. This is the current context for any new sessions (see below) that the user establishes.

A default context clause in which to interpret commands, including specific use and rename clauses to select objects and programs frequently used.
@End(SEnumerate)

@SubSection(Groups)
A group is a set of users. A user may belong to any number of different groups. Groups are used to provide aggregate access control, i.e. access can be granted to a group instead of to each of the individual members of the group.

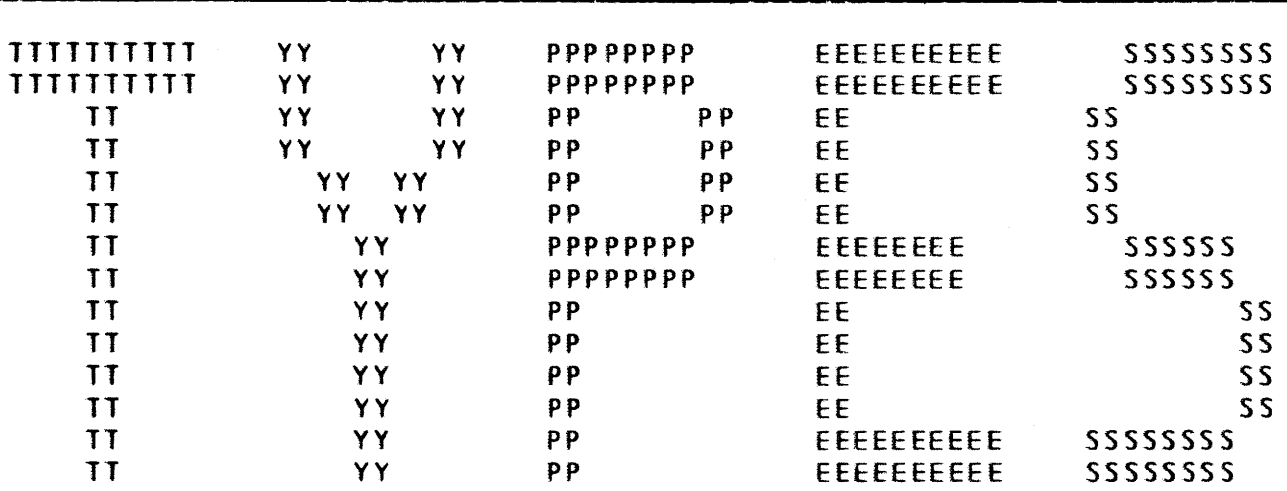
@SubSection(Session)
Session is a term that is used broadly to represent the contents of an interaction between a user and the environment. While active, the session acts for the user, providing the tasks necessary for user execution. Each session has a unique name, its session_id, that is attached to the base of each of the stacks of all of the tasks making up the session. This common identification is used to provide provide dyanamic inheritance of state between the tasks of the session. The inheritance is implemented by mapping session_id to interesting characteristics of the session (e.g. editor, terminal, user, group, current context). When needed, this information is extracted from the map using the session_id implicit in the task of the requestor. A number of commonly useful pieces of state are kept in these maps by the evironment; others can be added by new subsystems or users.

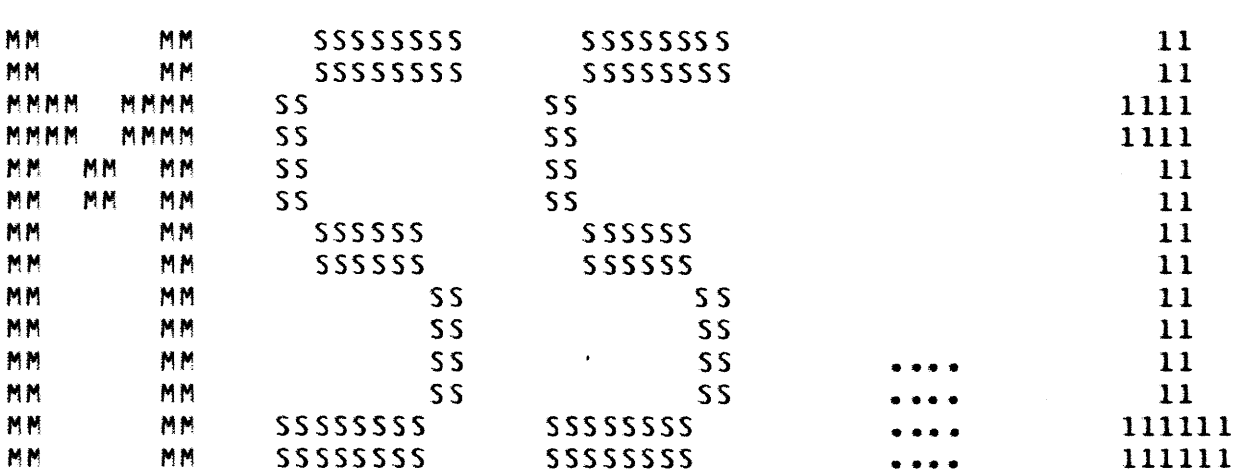
Sessions provide continuity from one period of interaction to another. When a session is inactivated, the environment saves characteristic information associated with the session_id. When the user resumes the session, this retained information provides continuity with the state prior to suspension.

@SubSection(Jobs)
A job is a logical thread of control as seen by a user. Although the job can contain an arbitrary number of tasks, it represents an autonomous entity started by the user to accomplish a purpose. Jobs form a subdivision of the session name space. This division makes it possible for different logical threads of control to have a common dynamic inheritance that is different from that provided for other jobs in the same session. Information associated with job_id (current source/destination of input/output, storage heap, file naming context, selection, etc.) is more execution-specific than that associated with session_id, but there is no hard distinction.

@SubSection(Login)
Login is how the user acquires a session. For all of the traditional reasons, login validates the user's right to use the system by requesting a password.

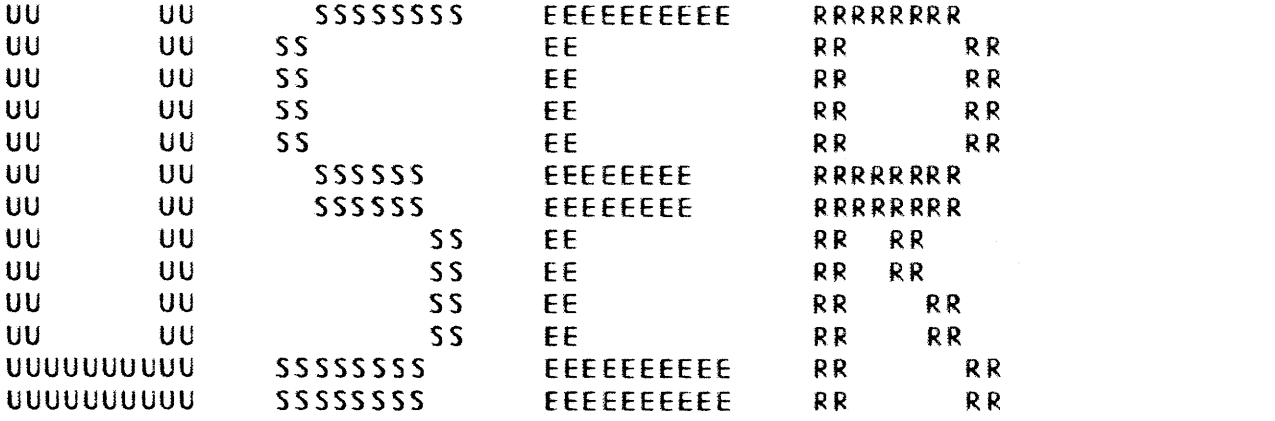
After validation, the user must establish which session is to be used (either by creating a new one or resuming an old one) and the type of terminal that is being used. Either or both of these could be chosen by apprpriate default.

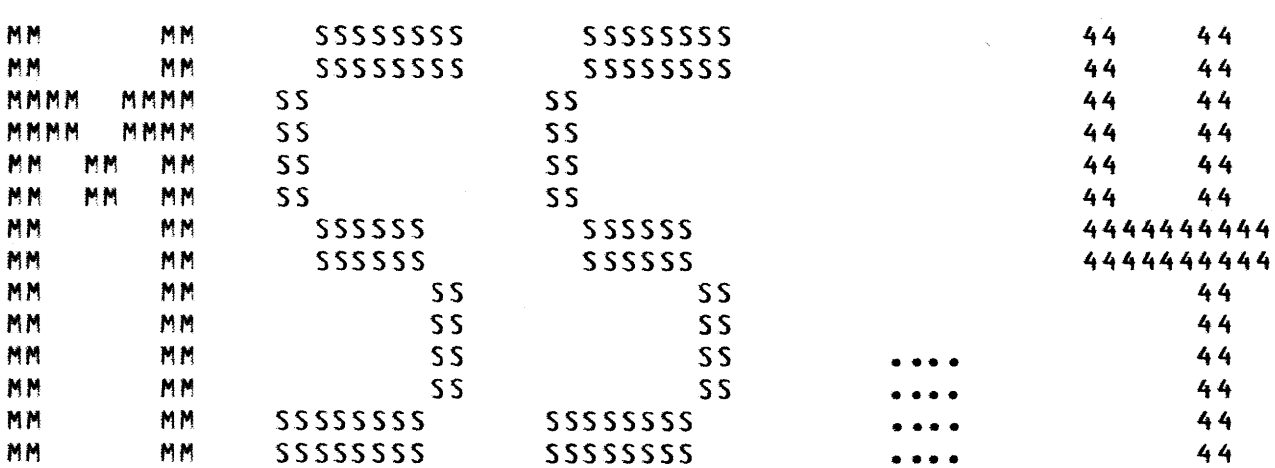




START Job INTERF Req #297 for EGB Date 15-Feb-84 12:18:08 Monitor: //, TOPS
File RM:<JIM.ENV>TYPES.MSS.1, created: 30-Nov-83 20:31:05
printed: 15-Feb-84 12:18:23
Job parameters: Request created:15-Feb-84 11:15:58 Page limit:63 Forms:NORMAL
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:ASC

@Section(System and Managed Types)





START Job INTERF Req #297 for EGB Date 15-Feb-84 12:18:08 Monitor: //, TOPS
File RM:<JIM.ENV>USER.MSS.49, created: 10-Feb-84 23:35:20
printed: 15-Feb-84 12:18:27
Job parameters: Request created:15-Feb-84 11:15:58 Page limit:63 Forms:NORMAL
File parameters: Copy: 1 of 1 Spacing:SINGLE File format:ASCII Print mode:ASC

@Part(UserInterface, root "[mtd.env]spec")
@Chapter(User Interface)
User interaction with the system and his own programs is through the editor. 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:

@Begin(Enumerate)
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 @I(Core Editor).

The type-specific, object-knowledgeable portion of the editor is called the @I(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.
@End(Enumerate)

@Section(Core Editor Concepts)
The Core Editor provides character editing facilities. This section is an attempt to define and briefly describe these.

@SubSection(Screen Structure)
A @I(Screen) is the entire contents of the display at a particular time. Screens are made up of opaque rectangular areas, called @I(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 @I(character positions) and, optionally, @I(borders). Borders are used to visually delineate windows. The character positions represent a bounded rectangular region of the quarter-plane of an @I(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 @I(word) is a portion of a line delimited by separator characters. Word boundaries are completely syntactic and are handled by the Core Editor.

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.

@I(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:
@Begin(SEnumerate)

A window containing the image of an object to be edited. This is called an @I(object window). It normally has top and side borders.

A @I(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. Although banners are implemented as windows, no editor operations will be provided initially for their manipulation.

A @I(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.

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.

The command window is an object window in its own right.
@End(SEnumerate)

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.

@SubSection(Cursors)
The physical screen has an apparent @I(cursor), marking the current position of the user's focus of attention (from the editor's point of view). This is called the @I(screen cursor).
If the cursor is within a window, it represents the:
@Begin(SEnumerate)
@I(Image cursor): (line, column) in the image on the window.

@I(Window cursor): (line, column) on the window.
@End(SEnumerate)
If the cursor is not within a window, the image and window cursors, and operations that depend on them, are undefined.

For each type of cursor, there are a variety of operations to specify its position. Changing on the position of one type of cursor often, but not always, changes the position of others.

The window and image cursors are closely linked. When they move in concert, the screen cursor moves across the window; when they move separately, the image scrolls on the window (in addition to possible screen cursor motion). The rest of this section deals with image and screen cursors and their relation to each other, ignoring window cursors to simplify the discussion.

Moving the image cursor causes the screen cursor to move. Moving the image cursor to a position that is currently not on the window causes the window to be scrolled. The screen cursor will not leave the current window because of an image cursor motion.

Moving the screen cursor causes the physical cursor to move without changing the image cursor. Having moved the screen cursor to a position within a window, any operation involving either the image cursor or the underlying image causes the image cursor for this window to be moved to the screen cursor.

Each window has a current image cursor position. Operations that change the focus to a previously visited image (and do not specify a particular position in that image) will place the cursor at the previous image cursor position. Thus, moving away from a window using screen cursors leaves the image cursor at the point of last interest rather than at the exit position.

A @I(mark) is a saved image position. Marks are stored in terms of absolute

image positions and do not change to adjust for inserted/deleted lines/characters.

@SubSection(Fonts and Designations)
Each character that appears in a window is displayed in some @I(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, etc.

Fonts are used to convey the usage of the characters displayed. In some cases the distinction is for user emphasis (e.g. keywords). More commonly, fonts are used impart a different meaning to the characters displayed. Each window has a default font. Characters that represent themselves and not otherwise special appear in this default font. There are, at least potentially, more different uses for fonts than a particular terminal supports. When this occurs, the same font will be used for more than one meaning, hopefully in a way that is not confusing.

@Paragraph(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.

@Paragraph(Selections)
Many editor operations require one or more implicit operands to accomplish the desired goal. The current cursor is one such implied operand; the current selection is another. Two kinds of selection are available: text and object. Text selections are formed by marking the first and last character positions to be selected, thereby selecting the text in between. Object selections are accomplished by various Object Editor operations. These operations select a region of the image that corresponds to a meaningful portion of the underlying object.

For either form of selection, the region of the image corresponding to the selection is presented in a font to provide visual feedback as to the extent of the selection. It is possible to convert object selections to text selections, so either type is acceptable to text operations. Text selections need not have any relation to object boundaries and are not appropriate for object operations. Even so, the font used for the two types of selection is typically the same, relying on the user to remember how the selection was formed.

@Paragraph(Designations)
A @I(designation) is one of three forms of meta-text that object editors can insert into an image to convey special meaning and support structured text within the editor paradigm. Designations are presented in non-standard fonts.

@I(Elision) is the process of removing detail from an image. The editor supports this by allowing a section of the object to be elided and represented by an @I(Ellipsis) mark (typically "...", but more meaningful phrases are possible). The ellipsis mark is presented in a special font and is treated specially in Core Editor operations. The ellipsis is a placeholder for the elided section of the image. As such, the Core Editor treats the entire ellipsis as a object, rather than as a collection of characters. Specifically, it is not possible to change individual characters. Moving or copying the ellipsis only moves or copies the underlying object if done by object operations.

A @I(Prompt) is a placeholder for an empty place in the object that the user

may want or need to fill. The prompt is an extension of the traditional notion of prompt as one or more characters printed at the beginning of a command line to signify readiness and remind the user of the program to which the command will be routed. Prompts are placed wherever the Object Editor expects the user to provide content. The prompt is printed in a distinguished font and disappears when any attempt is made to type over it. As a result, the prompt serves as a reminder and placeholder, but requires no effort to delete.

The contents of a prompt depends on the item to be entered and the amount of information that the underlying Object Editor has about reasonable values. The simplest form of prompt contains the name of the class of object that needs to be provided. For Ada, this would likely be a nonterminal in the abstract grammar, e.g. expression. In more semantically defined situations, the prompt might contain a reasonable initial value. The default value of a parameter or the default initialization of for a field in an aggregate are examples of prompts that, left alone, become the values provided. An operation is provided to convert the prompt text to plain text, allowing normal edit operations without losing the entire text of the prompt.

An @I(error) is a section of text marked by the object editor to indicate a problem of some sort. An error is treated as a prompt for editing purposes. Correcting the problem detected will cause the error to go away when the object editor re-formats the presentation of the object.

@SubSection(Mechanisms)
The following mechanisms are provided to support editing operations that are not primarily dependent on the apparent objects on the screen.

A @I(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 @I(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 @I(Yank Buffer) is a piece of an image that has been saved for later use, typically by a deletion operation.

An @I(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 @I(Stack) is a structure for saving a set of objects based on usage patterns. Stacks are used to store marks, windows, images, selections, yank buffer, and extended commands. The operations described below make it possible to cycle through the previous instances of each type in an orderly manner. The primitive operations are:
@Begin(SEnumerate)
Push. Add/move an item at the top of the stack.

Next. Examine the next item down the stack.

Previous. Examine the previous item up the stack.

Top. Examine the item at the top of the stack.
@End(SEnumerate)
Next (previous) "wraps" to the top (bottom) when applied to the bottom (top).

@Section(Object Editor)
The object editor provides the transformations between the object and its image. This is done by incremental parsing and pretty-printing operations. Four basic operations are supported for viewing and changing objects.
@Begin(Enumerate)
Display. Create the image of an object.

Format. Parse text changes made to the image into the object and update the image to reflect the changes. This provides an opportunity for incremental syntax checking and correction and pretty-printing.

Commit. The object is in a user-desired state. Take the appropriate actions to reflect this intent. For most object types, this means saving the object. For commands, it causes the command to be executed.

Revert. Bring the image back to the state it had following the last commit. This provides a coarse-grain undo facility.
@End(Enumerate)

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.

@SubSection(Pointing)
One of the basic notions of the environment is that objects are interconnected and that it is easier for the user to point at an object of interest and request information than it is to formulate a specific procedural request naming the object and the desired information. Having selected an object of interest (either explicitly or by simple cursor placement), at least the following broad categories of information can be requested:
@Begin(Description)
Definition@\Show the definition of this object. For a reference to an Ada object, this move the cursor to the declaration of the object. From the defining occurrence, it moves the cursor to the definition in the body or private part.

Completion@\Provide information about the possible correct completions for the object of interest. Fill out all or part of a name on the basis of a prefix or pattern. Fill out the remainder of a syntactic structure. Provide prompts and/or values based on the type of the object that will make it possible for the user to complete the object. An example of all of these would be entry of the prefix of a procedure name and having it complete to a procedure call with full named-parameter notation for the call prompt-designated presentations of the defaults and nonterminal prompts for parameters without default values.

Help@\Explain the object. As distinguished from definition, show a description of the object and its use. For an error, show an explanation of what was wrong, associated rules, etc.

Attributes@\Display attributes of the object that are not part of its image. Instances of this sort of information would be modification date, creator, and installation/elaboration status.
@End(Description)

@Section(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.

@SubSection(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.

@SubSection(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.

@Section(Program Execution)
@SubSection(Commands)
Programs and editor commands and Ada.

@SubSection(Jobs and Sessions)
What are jobs and sessions.

@SubSection(Context)
What is context, how is it used and set.

@Section(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.

@Paragraph(Types)
The set of object types is:
@Begin(SEnumerate)
Character cursor. Character insertion, image position and marks.

Command. Command window anc history.

Designation. Elisions, error and prompts.

Macro.

Line.

Screen cursor. Motion on the screen.

Selection. Both object and text selection.

Window.

Yank buffer.

@End(SEnumerate)

@Paragraph(Operations)
The following is a brief description of each of the classes of operations and the types that each applies to.
@Begin(Enumerate)
Planar movement (up, down, left, right)
@Begin(SEnumerate)
Cursor. Move user cursor on the image

Screen cursor. Move user cursor on screen.

Selection. Select parent, child or brothers.

Window. Scroll the window over the image.
@End(SEnumerate)

Relative positioning (next, previous, beginning_of, end_of)
@Begin(SEnumerate)
Designation.

Line.

Word.
@End(SEnumerate)

Modification. (copy, delete, insert, move, transpose; capitalize, lower-case, upper-case)
@Begin(SEnumerate)
Character.

Line.

Selection.

Word.
@End(SEnumerate)

Stack. (next, previous, push, top)
@Begin(SEnumerate)
Command. Manipulate history. Push is implied by execution.

Mark.

Selection.

Yank buffer.
@End(SEnumerate)

@End(Enumerate)

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

@Section(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.