Medley Interlisp:
The Interactive Programming Environment
(derived from Interlisp-D)
Version 1.0

by
Stephen H. Kaisler, D.Sc.

ACKNOWLEDGEMENT

The restoration of Interlisp and its rehosting on modern operating systems and computer systems has been a work in progress for over three years. Many of the original designers and authors of major sections of Interlisp-D have participated in this restoration project. The team working this restoration project includes:

Larry Masinter
Nick Briggs
Ron Kaplan
Arun Welch
Nick Stumbo
John Vittal
John Cowan
Michelle Denber
Abe Jellinek
Frank Halasz

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

My first book on Interlisp addressed the basic characteristics of Interlisp-10, a dialect of the Lisp programming language. Features which are common to most versions of Interlisp were described with numerous examples. In this volume, I explore the features of Interlisp-D: The Interactive Programming Environment. Interlisp-D was a rehosting of Interlisp to a new class of powerful, microprogrammed computer systems specifically designed to execute Lisp and other high level languages efficiently.

Authors Note

This volume was originally prepared in the mid-to-late 1980s when Xerox was manufacturing and selling D-machines. With the advent of Common Lisp and commodity-based workstations, the market for D-machines collapsed. Interlisp-D languished for quite a while although versions were developed for DEC's VAXEN running under VMS. In the late 2010s, a group of the original developers of Interlisp-10 and Interlisp-D decided to resurrect Interlisp-D and port it to a number of personal computer systems: Apple's Macs, Windows machines, and Linux machines.

When I joined the group, I decided to update this volume to be useful to Interlisp users. It follows the structure of the first volume, but has been edited to remove most of the references to Interlisp-D in the later chapters. Additionally, I have deprecated sections relevant to hardware resources which no longer exist – these have been marked by dashed lines before and after a section with the word DEPRECATED in the middle. These sections have been retained, however, for historical purposes.

This volume is entitled *Medley Interlisp: The Interactive Programming Environment,* because it is based on the Medley release of Interlisp-D.

After Section 1.1, I will now use the name Interlisp to refer to the rehosted system that runs on a variety of modern machines.

NOTE: Sections marked ----------Deprecated---------- refer to hardware, primarily printers, which are no longer manufactured by Xerox nor will be supported by Interlisp. They are included in this first edition of this book until software is developed to support modern printers. In succeeding editions, the deprecated sections will be removed.

1.1 INTERLISP-D: THE STORY CONTINUES...

Interlisp-D is a dialect of Lisp (Kaisler 1985) which incorporates an extensive set of facilities for enhancing program development including syntax extension, error correction, history retention, and source code analysis. Interlisp was originally developed on DECSystem-10 processors under the TENEX and TOPS-10/20 operating systems.

During the mid-70s, researchers at Xerox PARC realized that a more efficient execution engine would be required for the development of larger applications. An initial port, called AltoLisp, was developed for the Xerox Alto, a small personal computer. However, the Alto was limited in its available memory and disk storage because it was based on extensions to a general-purpose 16-bit computer (similar to the Data General Nova series).

Xerox PARC researchers concluded that new execution engines were required to support Interlisp in a personal computing environment. Thus, they developed the Dolphin (aka Xerox 1100 Scientific Information Processor), the Dandelion, and the Dorado (aka the Xerox 1108 and 1132 Artificial Intelligence Workstations, respectively) personal computers. In 1985, Xerox produced the next generation of customized Lisp processors, code-named Dove, which became the Xerox 1185/1186 Workstations. These workstations set new price/performance levels for personal Lisp workstations.

This new series of workstations extended the interactive programming environment pioneered by the Alto - namely, a bit-mapped screen with a pointing device, the mouse, and a window-oriented screen

management system. It allowed the Xerox researchers to extend the interactive programming paradigm through the use of graphic-oriented features.

During the rehosting process, a large part of Interlisp, which was written in DECSystem-10 assembly language, was rewritten in Interlisp itself. This activity was made possible by the fact that these processors, microprogrammed to efficiently execute Interlisp primitive functions (Lampson 1980), appeared to be Lisp direct execution machines. As a result, significant performance gains were realized as portions of the Interlisp system were recast in Interlisp-D.

1.2 OBJECTIVES OF THIS TEXT

This text should be viewed as a companion volume to the text ***Interlisp: The Language and Its Usage*** (Kaisler 1985).

The objective of this text is to describe the extended features of Interlisp-D which were developed for the display environment as implemented on the Xerox artificial intelligence workstations. In the same vein as my first volume on Interlisp, this text describes the functionality of Interlisp-D and provides numerous examples of how the functions work. This volume contains numerous figures which depict the results of the functions as they appear on the display screen.

However, this text cannot address all of the features of Interlisp-D due to limitations on the size of the text. Thus, I have had to omit discussion of the communications subsystems (which rightly belong in a book of their own), many of the Lisp library packages (such as File Browser), and some of the Lisp User packages. These have been moved to a separate forthcoming volume.

1.3 WHAT IS INTERACTIVE PROGRAMMING?

A major advance in computer science has been the development of software tools to assist in the programming process itself. The earliest tools were text editors, compilers, and debuggers. Later tools included source code analyzers, source code formatters, macro processors, and a variety of ther tools. It wasn't until recently that display technology became so relatively inexpensive that many organizations could afford to provide their programmers with high-resolution, bitmapped graphics personal workstations. But, it is just this trend which has again revolutionized the way we think about the programming process.

A collection of programming tools is often referred to as a *programming environment*. At its lowest level this is simply a set of computer-aided software design tools. While useful, they often don't work together well because each tool has no sense or cognizance of what other tools it can interact with. In fact, this is a common gripe associated with Unix systems because of the mode of communication enforced by Unix between different programs.

Interlisp, as I remarked in the previous volume, provides an *integrated* programming environment. It forges a strong coupling between different subsystems by virtue of the fact that all functions exist in the same name space. Thus, within limits, all functions, modules, and subsystems can know (but don't necessarily have to) about the other software in memory. Taking advantage of this feature, however, is what gives Interlisp much of its power.

With the advent of high-resolution, bitmapped workstations, the human interface to programming tools has been extended in such a way as to greatly enhance programmer productivity. Barstow et al [bars84] have defined *interactive programming environments* as having four features:

1. They provide a large set of programming tools within a unified framework, most of which are specific to a particular programming language.

2. They use to good advantage the fact that a program is more than just a string of characters, e.g., that it has an underlying (possibly deep) structure which can be used to organize programming tasks.
3. They support incremental program development for both design and maintenance.
4. They are highly interactive - often supporting multiple channels of high bandwidth between the user and the environment.

Interlisp provides the first three features. In fact, it is the only language that I know of that allows you to carry support for these features to the greatest possible extent. However, it evolved in a time-shared, hardcopy world. Interlisp satisfies the last feature through its support for high-resolution, bitmapped displays. But, vestiges of its earlier heritage still remain.

1.4 PHILOSOPHY

Computers play a major role in many aspects of our lives. The next generation of computers is expected to display capabilities corresponding to human-like reasoning with massive databases and versatile communications networks. Recent research in artificial intelligence programming indicates that the user interface will play a crucial part in the communication between man and machine. Because the interactive display environment supported by the Xerox Artificial Intelligence workstations is so powerful and flexible, it provides us with the oppurtunity to explore different paradigms of human-computer interaction.

Interactive systems that make it easy for man to use machines are often called ***user-friendly***. Moran [mora81] has identified the following attributes as characterizing a user-friendly system:

User-Friendly System Attributes

- Functional
- Easy to use
- Easy to learn
- Flexible
- Consistent
- Logical
- Natural
- Readily available help

A discipline known as ***human engineering*** has sprung up to address the issue of how to make machines, particularly computers, more amenable to human utilization. Computers are tools which help us to perform tasks just like many of the other implements that we encounter every day. However, computers are rather unique in that we communicate with them and they with us. In the following sections, I will examine some of the human engineering issues that impact upon the perceived friendliness of a user interface.

1.4.1 PSYCHOLOGICAL PRINCIPLES

The interface is the user's window to the capabilities of the underlying system. Through it, he perceives what the system can and cannot do for him. He also perceives how he can control the system in order to accomplish useful work. The user develops his own view of the system from what he sees through the user interface.

Schneiderman [schn80] has identified several major principles:

1. A user's short-term memory is very limited as demonstrated by Miller's seminal work where he deduced the magic number 7 plus or minus 2. The user's processing capacity is very small and in constant danger of overload. Interactive systems should be designed to allow the user the select the rate and amount at which information flows to him.
2. Humans like to control their environments. As humans gain more experience with computers, their desire for control increases. Interactive systems should be designed to

make the user feel that he is in control. As simple a matter as the wording of the prompt message has a significant psychological effect on the user.

3. Most people are subject to the phenomenon of closure - the feeling of relief when a task is completed and the relevant information is no longer needed. This phenomenon produces a desire to complete tasks and, thereby, reduce memory load. Interactive systems should be designed to permit problems to be partitioned so that closure can be often attained. That is, the user should be able to decompose the problem into subproblems each of which can be solved largely in its own right while still contributing to the total solution.
4. A user's attitude about a system has significant impact upon his learning and performance. Anxiety about using a system can reduce short-term memory capacity and inhibit performance. Interactive systems should be designed to make the user feel at ease. Reducing anxiety is often a function of ensuring consistency throughout the interface and coherency among the functions provided by the system.
5. Response time is a critical issue. In general, most users will prefer a response time which is a function of the action being taken. They also prefer minimal variation in the response time. Response time is often evaluated according to the perceived complexity of the activity being performed.
6. Errors have a significant impact on the user's perception of a system. Interactive systems should be designed to avoid (among others) errors due to information overload or inadequate instructions for non-routine tasks.

Each of these features is significant in that it impacts directly upon human productivity.

1.4.2 USER INTERFACE MODELS

Different categories of users will use an interactive programming environment in different ways. This is particularly true of commercial products where the manufacturer/vendor has no control over who has access to the system once it leaves the shop. At a simple level we might divide users into two categories (within the context of using Interlisp): novices and experts.

Novices are largely concerned with getting the task done. Using a computer is a problem solving activity, but the computer system itself may be the problem for the novice. He will usually solve the problem in the most expeditious way (and the simplest) without regard for efficiency. Novices are usually familiar enough with the system to use it to perform specific tasks. However, because they do not understand the actions associated with the results they achieve, they often cannot apply it to another purpose.

Experts, on the other hand, are skilled in interacting with the computer and view it as a routine cognitive skill. Indeed, to many expert programmers, it is a continuing challenge to determine how to accomplish tasks efficiently and accurately.

Both types of users are prone to experimentation. Thus, large behavioral differences fade with practice by the novice.

In order to satisfy both user categories, interactive systems need to be developed with multiple levels of interfaces. This permits novice users to communicate in a structured manner while permitting experts to employ well-known short-cuts for greater efficiency and productivity.

There are three basic interface models to serve the needs of users of different types:

- Menu-driven
- Fill-in-the-Blank
- Parameter-driven

Although any system could be accessed by just one of these models, for some functions/operations, it would not be practical. Substantial work has been devoted to developing the ideas that underlie these models. Each model has different factors that must be considered in the design of an interface.

1.4.2.1 Menu-Driven Interfaces

The menu-driven interface relies heavily upon a user's ability to recognize and respond to predefined prompts. Thus, little formal training is required and it provides a mechanism for initiating the user in the usage of the system as quickly as possible.

However, users are generally forced to follow predefined paths with little or no ability to backtrack if an erroneous selection is made. Among the factors to be considered are:

- Tree-structured or cascaded menus provide a natural mechanism for traversing a hierarchy of choices.
- The number of items in a menu should probably be no more than 9 (the magic number 7 plus 2). The number of menus should be no more than 5 (the magic number less 2). If more options/levels are required, the functionality should be decomposed further among different modules.
- Alphabetically organized menus can provide faster search because most people scan the menu from the top.
- Menu options should be task-oriented rather than position-oriented.

Common types of errors should be watched for:

1. Accessing the desired function in a non-optimal way (or so it seems to the user).
2. Taking the wrong path to the wrong function, where immediate steps establish system variable values that are incorrect for the task at hand.
3. Unclear wording of menu options or confusing menu option names.
4. Having difficulty in distinguishing the name of a command from the effect of its action.

1.4.2.2 Parametric Interfaces

Parametric interfaces are unstructured interfaces. The user issues a set of commands, including erroneous ones, which must be processed by the machine. Interactive systems need to provide the following capabilities:

1. They should be sensitive to errors, including the identification of incorrect syntax and the correction of same, and to the handling of errors.
2. They should inform the user when they don't understand him; preferably with varying degrees of verbosity.
3. They should be able to provide the user with information that explains what has happened.
4. A user should be able to personalize an interactive system to his own idiosyncrasies, especially when the system has some knowledge of the user's likes and dislikes.

Within this framework, mnemonic names are easiest to learn and remember. Commands and function names should naturally reflect their usage. Unfortunately, in the Lisp environment, some function names have a long and hoary history, but such universal acceptance, that changing them has long since been precluded.

Spelling correctors can assist users by fixing simple mistakes. Interlisp provides a powerful spelling corrector, but it is not universally used in all subsystems. Moreover, the algorithm has not been refined since the initial implementation under Interlisp-10 and appears in some instances to be slow and cumbersome.

1.4.3 PROTOTYPING

One of the major benefits of Interlisp is its support for rapid prototyping of both applications and user interfaces.

Interactive systems used to be designed inside-out, i.e, one built the basic functional components and then grafted a user interface onto them. This approach limited the functionality apparent to the user and impacted ease of use of the resulting system. Forcing the user to conform to the completed system was not a guarantee of success.

Prototyping is an iterative specification and design method. One of its goals is to provide a clear channel of communication between the end user and the designer by supporting the incremental evolution of the system specification in a way that leads to complete and unambiguous interpretation by the user. Essential to this process is the direct involvement of the end-user in the design process as well as the continuous feedback on the systems functionality. With the advent of high resolution, bitmapped workstations, prototyping has emphasized the visual aspects of the system often even more so than the internal functionality.

The benefits of this approach on the user interface include:

1. The user interface can be customized to multiple users to respond to different interaction styles.
2. The interface can support a wide variety of experience levels or task structures.
3. Different user interfaces can be evaluated to determine which enhances productivity the most.
4. Applications will appear more consistent to users if they share similar interfaces.
5. On-line assistance, both textual and graphical, will improve the user's learning curve.

1.4.4 ROLE OF INTERLISP

The problem of developing good user interfaces typifies the current software crisis. Hardware engineering has far surpassed software engineering in its capabilities. Similarly, software engineering has surpassed human factors engineering.

The development of good user interfaces is dependent upon the development of design tools which increase our understanding of the user's needs, support and expedite the design of user interface software, and minimize the software maintenance costs.

Interlisp provides a set of tools that support not only the development of good user interfaces but also the development of large software systems.

1.5 DIFFERENCES BETWEEN INTERLISP-D AND INTERLISP-10

Interlisp-D is largely compatible with Interlisp-10/VAX for most of the kernel functions and the basic subsystems (e.g., those not depending on the display environment features). It is much less compatible with Interlisp/370 which is a subset of Interlisp-10. As of this writing, Interlisp/370 is no longer maintained by the University of Uppsala. However, the author found it to be a good tool for initially learning about Interlisp in the absence of a D-machine.

Xerox's commitment to its Artificial Intelligence Workstations coupled with the impending demise of the DecSystem-10/20 technology meant it was no longer feasible to maintain two variants of Interlisp.

Because many of the kernel functions of Interlisp have been written in microcode during the rehosting, some kernel functions operate differently and some functions have been eliminated in their entirety. This section presents a brief survey of the differences in order to alert you to programming considerations when attempting to port source code from the Interlisp-10/VAX environment to the Interlisp-D environment.

This section follows the ***Interlisp Reference Manual*** [Xerox 83]. It also includes material from various editions of Masterscope, the Interlisp Users Group Newsletter, and the Interlisp Release Notes.

1.5.1 TYPE NUMBERS

Interlisp assigns type numbers to Interlisp objects in a different manner than Interlisp-10. Thus, the function **NTYP** has been eliminated. You should use the function **TYPENAME** to obtain implementation-independent type information.

Note: In Medley Interlisp, some of the Interlisp-D functions were eliminated.

1.5.2 ARITHMETIC

Arithmetic is significantly affected by the differences in the characteristics of the hardware on which the versions of Interlisp are implemented. There are significant differences between the DECSystem-10, the VAX-11/7xx, and the Xerox 11xx family of processors. Indeed, each of the Xerox 11xx processors is implemented in a different fashion. However, Interlisp has attempted to retain a uniform number representation across the processors.

The major differences between Interlisp-10 and Interlisp-D with respect to numbers are:

1. The small number range (i.e., that recognized by SMALLP) is [-65536, 65535];
2. The overall number range is smaller due to the lesser number of bits per word (e.g., 32 bits versus 36 bits);
3. The function **SETN** is treated as a **SETQ;**
4. The functions **OPENR** and **CLOSER** have been eliminated;
5. The functions **VAG** and **LOC** are defined as inverses, but arithmetic operations cannot be carried out on LOC values;
6. **FLTFMT** does not accept DECSystem-10 numeric floating point formats, but will accept formats that are acceptable to PRINTNUM;
7. **NUMFORMATCODE** is treated as a no-operation; and
8. Certain arguments to **FLTFMT** are ignored or interpreted differently.

1.5.3 VARIABLE BINDING

Interlisp-D uses the deep binding method for associating values with variables, whereas Interlisp-10 uses the shallow binding method.

Most Interlisp programmers should not notice any differences unless they are overly concerned about efficiency. If so, they should consider the following notes:

1 It is more efficient to pass information to functions as arguments rather than allowing it to be freely referenced.
2. Variables which are never bound in functions (i.e., whose top-level value is used only) should be declared as GLOBALVARS.
3. RESETVARS should be used judiciously to protect your environment.

1.6 OVERVIEW OF THE TEXT

This volume is oriented towards a thorough description of the features and capabilities of the Interlisp programming environment. Primarily, it discusses subsystems which are extensions to the basic Interlisp capabilities. The tools described in Part II are built upon these fundamental capabilities.

Chapter 1, where you are now, provides an overview of the language, a smattering of history, and an overview of the text. It also discusses the differences between Interlisp-10 and Interlisp. Because display-

oriented programming requires a different perspective on information manipulation and presentation, a philosophy of interactive programming is also discussed.

Chapter 2 describes the concepts behind the interactive programming environment as augmented by powerful, flexible bit-mapped displays which support multitasking. Because the Xerox workstations directly execute Interlisp code, the user is provided considerable control over the display screen, the keyboard, and the printer which would normally be mediated by the operating system in more complex systems. Fonts are discussed in this chapter because they seemed not to fit well with the material in other chapters.

I discussed the floppy disk subsystem in this chapter because it did not seem to fit in any other chapter in the original version of this volume. However, as floppy disks are no longer used in modern systems, this section will be deprecated.

Most users of Interlisp will immediately recognize that its primary form of access to other software is intended to be over a communications network like ARPANET.

Chapter 3 describes the Display Manager which is a subsystem of Interlisp. The display management subsystem is concerned with managing the display screen. This chapter describes the concepts underlying the display screen: positions, regions, and bitmaps. It describes how to manipulate bitmaps and display them upon the screen. It describes display streams which are the "channels" by which a program communicates with the user. Display stream are a specialized instance of the more general image streams which permit extensive graphics to be displayed. One of the features gained by bitmapped screens is the ability to utilize different character fonts to present textual information.

Chapter 4 describes the Input Management subsystem. Under input management the user can control the mouse and the cursor which provides the means for pointing and selecting objects on the display screen. A generalized input function, PROMPTFORWORD, is also described. PROMPTFORWORD is built upon the TTYIN subsystem which is discussed in Chapter 10. You may want to contrast this subsystem with ASKUSER which is described in [kais86].

Chapter 5 describes the Window Manager. The Window Manager is responsible for creating, destroying, and manipulating the windows displayed on the screen. Each window corresponds to a process which is being run. More than one window can be dedicated to the same function or multiple functions can interact with the user through one window. Learning effective use of the window management system is crucial to developing successful user interfaces in the interactive programming environment.

Chapter 6 describes the Menu Management Package. Menus provide a mechanism for displaying the choices available at a particular stage of the processing. A user may choose something from the menu by moving the cursor to the item and clicking the mouse button. So, rather than typing long commands or file names, you merely select commands or options from a menu by a "point-and-click" paradigm. This paradigm is an extremely powerful one because it opens up a wide variety of options which can be understood with a few glances at some easy-to-read menus.

Chapter 7 describes image streams. Image streams are a generalized display mechanism for displaying both graphics and text. Image streams provide an object-oriented paradigm for manipulating the information to be displayed to the used in a window. Originally, image streams were intended as a mechanism for inserting graphical objects into textual files (a la an integration with TEdit text files). However, image streams have proven to have a much more general applicability by providing a generalized object-oriented interface.

CONVENTION: I have used a different style of formatting Interlisp code than one would normally see when pretty-printing in order to make the function definitions and code fragments easily readable.

1.7 UPDATES TO THIS TEXT

As this restoration project proceeds, some enhancements and updates to Medley Interlisp will be made to support its modernization and further development as a modern programming environment. These updates and enhancements will be reflected in updates to this volume.

2. THE INTERACTIVE PROGRAMMING ENVIRONMENT

Interlisp is distinguished from Interlisp-10/VAX/370 by its incorporation of a ***display environment***. Interlisp emphasizes the use of interactive graphics as a mechanism for user-computer communication. Rather than typing large amounts of information into a program, the user may select objects displayed on the screen or choices from a menu.

Interlisp runs on a variety of workstations including the venerable Xerox 1100s. Most of the examples provided in this book were run on a Xerox 1186 with 3.7 MBytes of memory, a 19 inch display screen, and a 4045 Laser printer (which hardly ever worked correctly). In 1987, Xerox indicated that it would port components of XAIE (the Xerox Artificial Intelligence Environment, which includes CommonLisp) to Sun Microsystem's workstations based on the SPARC chip. No commitment has been made to prting the entire Interlisp environment to a SPARC-based system at this time.

This chapter describes some of the hardware functions which have been incorporated into Interlisp. These functions include those corresponding to the video screen, the floppy disk, and some of the printers. Function concerned with mouse handling are included in Chapter 4 because they are more appropriately discussed with input functions.

2.1 FONTS

One of the nice features which Xerox pioneered (and which every other manufacturer has since adopted) is the ability to manipulate multiple fonts in text and source code. This feature existed to a limited extent in Interlisp-10. It is more pronounced when it is used in Interlisp, expecially when you see the font changes at your terminal before you print your document.

A *font* is a specification for the way a character will appear when displayed on an appropriate display medium (whether screen or paper). Fonts that are used to display characters on a screen are called DISPLAYFONTS, while those used to display characters on paper were called INTERPRESSFONTS or PRESSFONTS. Fonts are defined by three key characteristics:

FAMILY	The distinctive style or appearance
SIZE	The number of points used for display
FACE	The appearance of the font

Interlisp supports a large number of font families including Helvetica, Gacha, Elite, TimesRoman, and OldEnglish. These families are supported in a variety of sizes ranging from 8 points up to 72 points.

The *face* of a font governs its appearance. Face is specified as a three-element list consisting of:

WEIGHT	The thickness of the characters. Options are BOLD, MEDIUM, and LIGHT.
SLOPE	The alignment of the characters. Options are *ITALIC* or REGULAR.
EXPANSION	The spread of the characters. Options are REGULAR COMPRESSED, and EXPANDED.

For convenience, the face may be specified as a three-character acronym for the corresponding list. Thus, MRR represents the list (MEDIUM REGULAR REGULAR). Also, certain common faces have been given names, including:

STANDARD	(MEDIUM REGULAR REGULAR)
ITALIC	(MEDIUM ITALIC REGULAR)
BOLD	(BOLD REGULAR REGULAR)
BOLDITALIC	(BOLD ITALIC REGULAR)

Fonts may also have *rotation* which indicates their orientation on the screen or page. A font in which characters are printed horizontally on a page has a rotation of 0. When characters are printed vertically in a column, the font has a rotation of 90 degrees.

Note: INTERPRESSFONTS and PRESSFONTS were specific to Xerox printers. Since these devices are no longer manufactured, the names are deprecated.

2.1.1 THE DESCRIPTION OF A FONT

A *font* is represented in memory as an object, called a *font descriptor*, which is addressed by a *font descriptor handle*. Font objects are created by the function **FONTCREATE**. The major fields and data types of a font object are presented in Table 2-1.

Table 2-1. Major Fields and Datatypes

Field	Description
FONTEXTRAFIELD2	POINTER
FONTCHARSETVECTOR	A vector of the characters that comprise the character set of the font.
FONTIMAGEWIDTHS	The image width of the font which is used by the IMAGEOPS routines
FONTAVGCHARWIDTH	It is used by DSPFONT to adjust the line length. It is expressed as a number of pixels.
FONTSCALE	POINTER
OTHERDEVICEFONTPROPS	POINTER
FONTDEVICESPEC	Contained the font specification required for specific devices, if coercion has been done. For example, a device specification for the display would appear as (GACHA 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY).
\SFRWidths	POINTER
\SFLKerns	POINTERS
\SFFACECODE	BITS 8
FBBDY	SIGNEDWORD
FBBDX	SIGNEDWORD
FBBOY	SIGNEDWORD
FBBOX	SIGNEDWORD
ROTATION	WORD
\SFHeight	WORD
\SFDescent	WORD
\SFAscent	WORD
LASTCHAR	The character code of the last character in the font.
FIRSTCHAR	The character code of the first character in the font.
\SFWidthsY	POINTER
\SFOffsets	The offset of each character in the image bitmap.
\SFWidths	An array of the advance width of each character, indexed by the character code, which is used for string width calculations.
FONTFACE	POINTER
FONTSIZE	POINTER
FONTFAMILY	POINTER
CHARACTERBITMAP	A bitmap containing the character images for the font; it is indexed by \SFOffsets.
FONTDEVICE	POINTER

2.1.1.1 DISPLAYFONTS

Displayfonts required files that contained bitmaps used to print each character on a screen. The files had the extension ".DISPLAYFONT". The file name specified the font style and size. For example, HELVETICA10.DISPLAYFONT contained the bitmaps for the font family Helvetica in size 12 points.

These files should be located on the hard disk of the machine on which Interlisp is installed as they will be used frequently. The directory in which these font files are loaded should be one of the values of variable DISPLAYFONTDIRECTORIES.

2.1.2 CREATING A FONT

You may create a new font descriptor using the function **FONTCREATE**:

Function:	FONTCREATE
# Arguments:	7
Arguments:	1) FAMILY, the font family name
	2) SIZE, the size of the font in points
	3) FACE, the face of the family
	4) ROTATION, the orientation of the font
	5) DEVICE, the output device for the font
	6) NOERRORFLG, a flag for errors
	7) CHARSET, a character set
Value:	A font descriptor handle.

A *font descriptor* specifies the information necessary to display the font on the specified device. Thus, you can think of the process of creating a font descriptor as one of customizing a font. Consider the following examples:

```
<-(FONTCREATE 'HELVETICA 12 'MRR)
{FONTDESCRIPTOR}#70,171260
```

If the font descriptor already exists, Interlisp merely returns the handle of the font descriptor. If the font descriptor does not exist, Interlisp reads the information describing the font from a font file which must be accessible on your local disk or via the network. If an appropriate font file is found for the specified device, it is read into the font descriptor object. If no file is found, Interlisp attempts to "fake" the font by looking for a font of lesser size and face information and modifying its parameters. Interlisp only implements rotations of 0, 90, and 270 degrees. Other values will cause an error to occur.

If no acceptable font file is found, Interlisp uses the value of NOERRORFLG to determine what to do. If NOERRORFLG is NIL, it emits an error message:

```
<-(FONTCREATE 'GACHA 96 'MIR)
FONT NOT FOUND
(GACHA 96 (MEDIUM ITALIC REGULAR) 0 DISPLAY)
```

Otherwise, it returns NIL. If an error occurs, the font descriptor is not created.
Note that a value of 0 for the font size is an invalid argument:

```
<-(FONTCREATE 'GACHA 0 'MRR)
ILLEGAL ARG
0
```

2.1.2.1 Structure of a Font Object

The structure of a font object is (for the Helvetica-10 font) presented in Table 2-x.

Tale 2-2. Font Object Structure

Field	Value
FONTEXTRAFIELD2	NIL
FONTEXTRAFIELD2	NIL
FONTCHARSETVECTOR	A handle
FONTIMAGEWIDTHS	NIL
FONTAVGCHARWIDTH	9
FONTEXTRAFIELD2	NIL
FONTCHARSETVECTOR	A handle
FONTIMAGEWIDTHS	NIL
FONTAVGCHARWIDTH	9
FONTSCALE	NIL
FONTEXTRAFIELD2	NIL
FONTCHARSETVECTOR	A handle
FONTIMAGEWIDTHS	NIL
FONTAVGCHARWIDTH	9
FONTSCALE	NIL
OTHERDEVICEFONTPROPS	NIL
FONTDEVICESPEC	(HELVETICA 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY)
\SFRWidths	NIL
\SFFACECODE	0
FBBDY	0
FBBDX	0
FBBOY	0
FBBOX	0
ROTATION	0
\SFHeight	12
\SFDescent	2
\SFAscent	10
LASTCHAR	0
FIRSTCHAR	0
\SFWidthsY	NIL
\SFOffsets	NIL
\SFWidths	NIL
FONTFACE	(MEDIUM REGULAR REGULAR)
FONTSIZE	10
FONTFAMILY	HELVETICA
CHARACTERBITMAP	NIL
FONTDEVICE	DISPLAY

2.1.3 TESTING FOR A FONT DESRIPTOR

You may test whether or not an arbitrary Interlisp object is a font descriptor using the function **FONTP**:

Function: FONTP
Arguments: 1
Arguments: 1) X, an arbitrary object
Value: X, if it is a font descriptor; otherwise, NIL.

Consider the following example:

```
<-(SETQ x (FONTCREATE 'GACHA 10 'MRR))
{FONTDESCRIPTOR}#70,171670

<-(FONTP x)
{FONTDESCRIPTOR}#70,171670
```

2.1.4 ACCESSING THE FONT PROPERTIES

The font descriptor stores the values of properties used to control the display of information for the particular device. You may access the font properties using the function **FONTPROP**:

Function:	FONTPROP
# Arguments:	2
Arguments:	1) FONT, a font descriptor handle
	2) PROP, a property name
Value:	The value of the font property

FONTPROP returns the value of the font property, if it exists. The following properties are currently accepted by FONTPROP, as presented in Table 2-3.

Table 2-3. Font Properties

Property	Description
FAMILY	The style of the font.
SIZE	A positive integer specifying the points.
WEIGHT	The thickness of the characters.
SLOPE	The alignment of the characters at the leading edge.
EXPANSION	The extent to which the characters are spread out.
FACE	A three-element list giving the typeface parameters.
ROTATION	An integer giving the orientation of the characters on the page.
DEVICE	The device that the font can be printed on.
ASCENT	The maximum height of any character in the font from the baseline.
DESCENT	The maximum depth of any character in the font below the baseline.
HEIGHT	The height of the character; equal to ASCENT+DESCENT.
SPEC	A quintuple of (FAMILY SIZE FACE ROTATION DEVICE) by which the font is known to Interlisp.
DEVICESPEC	A quintuple of (FAMILY SIZE FACE ROTATION DEVICE) describing how the font is represented on the device. It differs only if the font is coerced to an approximate one which actually exists on the device.
SCALE	The units per printer's point in which the font is measured.

Consider the following examples (where x is set as above):

```
<-(FONTPROP x 'FAMILY)
GACHA

<-(FONTPROP x 'SLOPE)
REGULAR

<-(FONTPROP x 'DEVICE)
DISPLAY

<-(FONTPROP x 'HEIGHT)
```

12

```
<-(FONTPROP x 'SPEC)
(GACHA 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY)
```

Note that Interlisp does not allow you to set the font descriptor properties because you may inadvertently set a system font property. FONTPROP is defined as a macro with the following definition:

```
(ARGS
   (SELECTQ
         (AND      (EQ (CAADR ARGS) (QUOTE QUOTE))
                   (CADADR ARGS))
         (ASCENT
                   (LIST (QUOTE FONTASCENT) (CAR ARGS)))
         (DESCENT
                   (LIST (QUOTE FONDESCENT) (CAR ARGS)))
         (HEIGHT
                   (LIST (QUOTE FONTHEIGHT) (CAR ARGS)))
         ... and so forth
   ))
```

2.1.5 COPYING A FONT

Once you have created a font descriptor, you may want to create simple variations of some of its properties to enhance the presentability of the information to be displayed. By copying the font descriptor, you can modify the values of certain font properties. You may copy a font descriptor using the function **FONTCOPY**:

Function:	FONTCOPY
# Arguments:	1-N
Arguments:	1) OLDFONT, a font descriptor handle
	2) PROP, a property name
	3) VAL, a new value for PROP
	4-N) PROP - VAL pairs
Value:	A new font descriptor handle.

FONTCOPY is a Nospread function. FONTCOPY returns a font descriptor which is a copy of the font OLDFONT, but differs in the values of the specified properties. Consider the following examples:

```
<-(SETQ hel12std (FONTCREATE 'HELVETICA 12 'MRR))
{FONTDESCRIPTOR}#56,45614

<-(SETQ hel24bold (FONTCOPY hel12std 'WEIGHT 'BOLD 'SIZE '24))
{FONTDESCRIPTOR}#56,45464

<-(FONTPROP hel24bold 'WEIGHT)
BOLD

<-(FONTPROP hel24bold 'SIZE)
24
```

The property names which may be specified for FONTCOPY are exactly those which are acceptable to FONTPROP. The first property may be a list rather than an atom. Consider the following example:

```
<-(SETQ hel14bold (FONTCOPY hel12std '(WEIGHT BOLD SIZE 14)))
{FONTDESCRIPTOR}#56,45540
```

```
<-(FONTPROP hel14bold 'WEIGHT)
BOLD

<-(FONTPROP hel14bold 'SIZE)
14
```

FONTCOPY accepts the property NOERROR which determines how cases where fonts cannot be created are processed. Its result is similar to NOERRORFLG in FONTCREATE.

2.1.6 DETERMINING THE AVAILABLE FONTS

You may determine the fonts which have been created for a particular font family using the function **FONTSAVAILABLE**:

Function:	FONTSAVAILABLE
# Arguments:	6
Arguments:	1) FAMILY, a font family name
	2) SIZE, the size of the font
	3) FACE, the face specification
	4) ROTATION, the orientation of the font
	5) DEVICE, the device on which the font is to be displayed
	6) CHECKFILESTOO?, a flag
Value:	A list of font specifications.

FONTSAVAILABLE returns a list of the fonts that match the given specification. Each of the first five arguments may have any of the values acceptable to FONTCREATE. They also may take the value * which indicates that any matches for that property should be reported. Consider the following examples:

```
<-(FONTSAVAILABLE)
((GACHA 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY))

<-(FONTSAVAILABLE 'HELVETICA '* 'MRR)
((HELVETICA 18 (MEDIUM REGULAR REGULAR) 0 DISPLAY)
 (HELVETICA 8  (MEDIUM REGULAR REGULAR) 0 DISPLAY)
 (HELVETICA 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY))

<-(FONTSAVAILABLE 'GACHA '* '* NIL 'DISPLAY)
((GACHA 8 (MEDIUM REGULAR REGULAR) 0 DISPLAY)
 (GACHA 10 (MEDIUM REGULAR REGULAR) 0 DISPLAY)
 (GACHA 12 (MEDIUM REGULAR REGULAR) 0 DISPLAY)
 (GACHA 10 (BOLD REGULAR REGULAR) 0 DISPLAY)
```

At a minimum, you must specify name for a specific font. If FONTSAVAILABLE cannot find any descriptors that match the specification, it returns NIL.

```
<-(FONTSAVAILABLE 'GACHA NIL)
ILLEGAL ARG
NIL

<-(FONTSAVAILABLE 'GACHA)
ILLEGAL ARG
NIL
```

When CHECKFILESTOO? is NIL (as in the above examples), only the font descriptors representing fonts loaded into virtual memory are examined. The value of the variable \FONTSINCORE is a list of the fonts loaded into virtual memory.

If CHECKFILESTOO? is non-NIL, the font directories for the specified device will also be searched. During file searches, the ROTATION argument will be ignored.

FONTSAVAILABLE checks not only the local disk, but also the font directories available over the network (if your machine is attached to a network).

----------Deprecated----------

2.1.6.1 Handling of PRESS Fonts

Press fonts are handled differently in Interlisp and in FONTS.WIDTH. The font widths for larger font sizes are scaled versions of the smallest font/face. Thus, FONTS.WIDTH does not store information about each instance, but dynamically scales it when the font descriptor is created. When FONTSAVAILABLE is called with CHECKFILESTOO? having the value T, it will find relative fonts whose size is indicated as zero. Consider the following example:

```
<-(FONTSAVAILABLE 'GACHA '* '* 0 'PRESS T)
((GACHA 0 (BOLD ITALIC REGULAR) 0 PRESS)
 (GACHA 0 (BOLD REGULAR REGULAR) 0 PRESS)
 (GACHA 0 (MEDIUM ITALIC REGULAR) 0 PRESS)
 (GACHA 0 (MEDIUM ITALIC REGULAR) 0 PRESS))
```

2.1.7 SETTING THE FONT DESCRIPTOR

While Interlisp supports a large number of fonts, you may require unusual characteristics for a particular problem. You can simulate an unavailable font by associating a font name with a particular quintuple of characteristics using **SETFONTDESCRIPTOR**:

Function:	SETFONTDESCRIPTOR
# Arguments:	6
Arguments:	1) FAMILY, a font family name
	2) SIZE, the size of the font
	3) FACE, a face specification
	4) ROTATION, an orientation
	5) DEVICE, a device name
	6) FONT, a font descriptor
Value:	A font descriptor.

SETFONTDESCRIPTOR associates the font descriptor with the set of characteristics given by the first five arguments. In effect, FONT may not exist, but is simulated by the characteristics. Consider the following example:

```
<-(SETQ     MYSPECFONT
           (SETFONTDESCRIPTOR      'GACHA 96
                                   '(MEDIUM REGULAR REGULAR)
                                   0
                                   'DISPLAY
                                   DEFAULTFONT))
{FONTDESCRIPTOR}#70,171670
```

Thus, whenever you try to print something on the display using MYSPECFONT, the value of DEFAULTFONT at the time it was set up will be used instead.

2.1.8 SETTING THE DEFAULT FONT FOR A DEVICE

Each device may have a default font which is used to display information if a font is not explicitly specified for the image stream associated with the device. **DEFAULTFONT** allows you to obtain the font descriptor which is the default font for the given device. It takes the form:

Function:	DEFAULTFONT
# Arguments:	2
Arguments:	1) DEVICE, a device name
	2) FONT, a font descriptor
Value:	The font descriptor associated with the device.

DEFAULTFONT returns the default font associated with the image stream type device, if FONT is NIL. Consider the following example:

```
<-(DEFAULTFONT 'DISPLAY)
{FONTDESCRIPTOR}#70,171670
```

If FONT is non-NIL and a font descriptor, it is set as the default font of the device. Consider the following example:

```
<-(DEFAULTFONT 'DISPLAY (FONTCREATE 'HELVETICA 10 'MIR))
{FONTDESCRIPTOR}#70,171670
```

which returns the previous font associated with the device.

We may define a function which allows you to print with a specific font as follows:

```
<-(DEFINEQ      (PRINT.USING.FONT (DEVICE NEWFONT MSG)
    (PROG       (OLDFONT)
                (SETQ OLDFONT (DEFAULTFONT DEVICE NEWFONT))
                (PRINT MSG IOSTREAM)
                (DEFAULTFONT DEVICE OLDFONT)
     ))
(PRINT.USING.FONT)
```

This function switches to the new font in order to display the message and then returns to the previous font.

2.1.9 ACCESSING CHARACTER BITMAPS

Each display font contains bitmaps for each character. You may access and modify the bitmaps for individual characters using the following functions.

2.1.9.1 Getting a Character's Bitmap

You may obtain the bitmap handle for a character in a font using **GETCHARBITMAP**:

Function:	GETCHARBITMAP
# Arguments:	2
Arguments:	1) CHARCODE, an integer representing the character or the atom representing the character

	2) FONT, a font descriptor
Value:	The bitmap handle.

Consider the following examples (where X is set as in Section 2.1.3 above):

```
<-(GETCHARBITMAP 14 x)
{BITMAP}#77,110700

<-(GETCHARBITMAP 'X x)
{BITMAP}#57,707
```

You may then examine the bitmap representing the character suing the Inspector or EDITBM. For example, the following function call produces Figure 2.1.

```
<-(INSPECT (GETCHARBITMAP 'X x))
{PROCESS}#71,222200
```

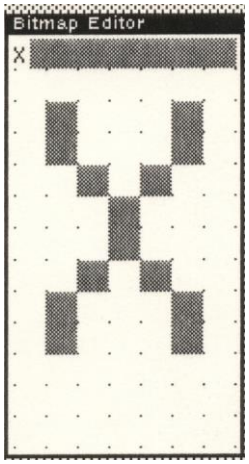

Figure 2.1 Result of (INSPECT (GETCHARBITMAP 'X x))

2.1.9.2 Changing the Bitmap of a Character

You may change the bitmap for a character in a display font using the function PUTCHARBITMAP:

Function:	PUTCHARBITMAP
# Arguments:	4
Arguments:	1) CHARCODE, an integer representing a character
	2) FONT, a font descriptor
	3) NEWCHARBITMAP, the new bitmap handle
	4) NEWCHARDESCENT, the depth below the baseline
Value:	The new bit map handle.

This function changes the bitmap of the character identified by CHARCODE to that given by NEWCHARBITMAP. If NEWCHARDESCENT is non-NIL, the descent of the character assumes that value.

The maximum descent of all characters in the font is modified if NEWCHARDESCENT exceeds the previous value. PUTCHARBITMAP allows you to define printing bitmaps for characters which might normally have unprintable representations for a particular display font. Consider the following examples:

You may also define your own fonts by creating bitmaps (using EDITCHAR) which are assigned to specific characters in a font.

Suppose I wanted to replace spaces by the character b/. I would create a new bitmap (see Figure 2.2) which represents this character.

```
<-(SETQ SPACEBM (GETCHARBITMAP 32))
{BITMAP}#55,3022

<-(SETQ NEWSPACEBM (EDITBM SPACEBM))
{BITMAP}#55,3000
```

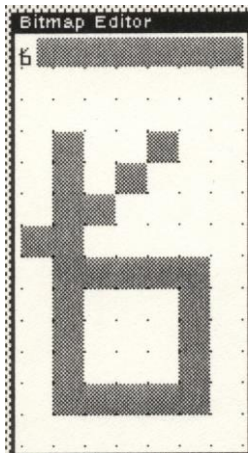

Figure 2.2 New Bit Map for Space

We can replace the bitmap for the character via:

```
<-(PUTCHARBITMAP 32 DEFAULTFONT NEWSPACEBM)
{BITMAP}#55,3000
```

Note: You must specify the font in which the character bitmap is to be replaced.

Figure 2.3 depicts a message printed using the new bitmap. Note that it is a good idea to save the old bitmap so that you may later restore it.

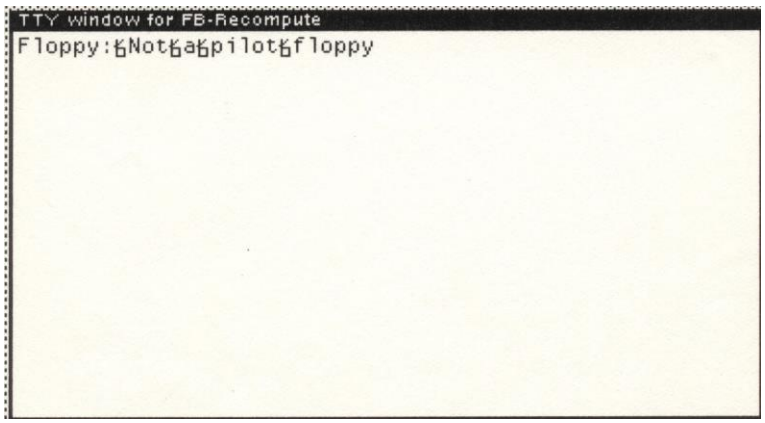

Figure 2.3 New Bitmap for Space

2.1.9.3 Editing Character Bitmaps

You may edit a character bitmap for a character of a font using **EDITCHAR**:

Function:	EDITCHAR
# Arguments:	2
Arguments:	1) CHARCODE, an integer representing a character 2) FONT, a font descriptor
Value:	A bitmap handle.

EDITCHAR invokes the bitmap editor on the bitmap of the specified character. Consider the following example (where X was set in Section 2.1.3 above) produces Figure 2.4.

```
<-(EDITCHAR 'Z x)
{BITMAP}#57,3124
```

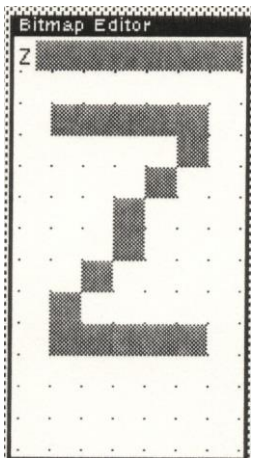

Figure 2.4 Result of (EDITCHAR 'Z x)

CHARCODE may also be an atom or a string, whence the first character is used to identify the character.

2.1.9.4 Displaying the Character Set

The interpretation of a character set depends upon the fonts that you have available at your workstation. You may determine the interpretation of character sets using the function **SHOWCS** which has been provided by Christopher Lane on the Info-1100 Bulletin Board. It takes the form:

Function:	SHOWCS
# Arguments:	2
Arguments:	1) CS, a character set index
	2) FONT, a font descriptor
Result:	A set of character codes.

SHOWCS displays in a window the interpretations of all character codes according to the specified font. SHOWCS may be defined as follows:

```
(DEFINEQ        (SHOWCS (LAMBDA (cs font)
    (LET        ((stream (OPENTEXTSTREAM NIL (CREATEW)))
                  (DSPFONT FONT STREAM)
                (from 0 to 255 as BYTE from (LSH cs 8)
                do
                    (BOUT STREAM BYTE)
                ))
    ))
```

2.1.10 FONT FILES AND FONT DIRECTORIES

Each font that is available to Interlisp is described by a *font file* which contains descriptive information about the font and the character bitmaps for each character in the font. The file FONT.WIDTHS contains information about the widths of characters for the Press fonts.

Font files are read by FONTCREATE when it builds a font descriptor for a font specification. FONTCREATE uses information in the font file to initialize the fields of the font object.

Each device type has a different set of fonts associated with it. The names, formats, and search strategies for locating fonts for a device differ with each device. The following variables, associated with the different device types, determine the directories that are searched for font files:

2.1.10.1 Display Font Directories

DISPLAYFONTDIRECTORIES is a system variable whose value is a list of directories that are searched to find font bitmap files for display fonts. Its initial value is:

```
<-DISPLAYFONTDIRECTORIES
({DSK})
```

You may reset the value to the disk directory where your fonts are stored. In many systems which are supported by a file server on a local area network, the value of DISPLAYFONTDIRECTORIES is a directory name on the file server host.

2.1.10.2 Display Font Extensions

DISPLAYFONTEXTENSIONS is a system variable whose value is a list of file extensions that are used when searching DISPLAYFONTDIRECTORIES. Its initial value is (DISPLAYFONT).

----------Deprecated----------

InterPress Font Directories

INTERPRESSFONTDIRECTORIES is a system variable whose value is a list of directories that are searched to find font bitmap files for display fonts. Its initial value is:

```
<-INTERPRESSFONTDIRECTORIES
({DSK})
```

You may reset the value to the disk directory where your fonts are stored. In many systems which are supported by a file server on a local area network, the value of INTERPRESSFONTDIRECTORIES is a directory name on the file server host.

Press Font Files

PRESSFONTWIDTHSFILES is a list of files, not directories, that are searched to find the font widths files for Press fonts. Press font widths are packed into large files (usually named FONT.WIDTHS). Its initial value is:

```
<-PRESSFONTSWIDTHS
({DSK}FONT.WIDTHS)
```

----------Deprecated----------

All of these variables must be set before Interlisp can perform automatic loading of font files.

2.1.11 FONT PROFILES

A *font profile* is a variable whose value describes the fonts to be used in printing different classes of expressions. Font profiles are used by PRETTYPRINT to improve the presentation of printed or displayed material.

Font changes in a file are signaled by the presence of a user-defined *escape sequence*, which is the value of the system variable FONTESCAPECHAR, followed by a character code. The character code specifies which font to use, e.g., ^A represents the first font and so on. The initial value of FONTESCAPECHAR is:

```
<-FONTESCAPECHAR
^|F
```

By inspecting some of the files on your system using TEdit, you can see how PRETTYPRINT inserts font change specifications.

2.1.11.1 Description of a Font Profile

The font profile consists of a list of the form:

(<fontclass> <font#> <displayfont> <pressfont> <interpressfont>)

<fontclass> is the font class name. <font#> is the font number for that class. For each font class name, the escape sequence consists of the font escape character followed by the character code for the font number. Thus, ^A for font number 1, ^B for font number 2, and so on. <displayfont>, <pressfont>, and <interpressfont> are font specifications for the fonts to be used when printing to the display screen, Press printers, and Interpress printers, respectively.

If <font#> is NIL for any font class, the DEFAULTFONT will be used, such as:

(SYSTEMFONT)

<font#> may also be the name of a previously defined font class, such as:

(LAMBDAFONT BIGFONT)

The DEFAULTFONT must always be defined. The font classes are presented in Table 2-4.

Table 2-4. Font Classes

Font Class	Description
LAMBDAFONT	Used to print the name of a function before the actual definition.
CLISPFONT	Used to print any CLISP words (atoms with the property CLISPWORD) if CLISPFLG is non-NIL.
COMMENTFONT	Used to print comments.
USERFONT	Used to print the name of any function in the file occurring within the body of a function or any function on the list FONTFNS.
SYSTEMFONT	Used to print any other defined function (usually external to the file).
CHANGEFONT	Used to highlight any changes to the file that have been marked by the Editor.
PRETTYCOMFONT	Used to print the operands of a file package command.
DEFAULTFONT	Used to print everything else.

The system variable FONTPROFILE is used to store the current font profile. Its initial value is:

```
<-FONTPROFILE
((DEFAULTFONT 1
        (TITAN 10)
        (TITAN 10)
        (TITAN 10))
 (BOLDFONT 2
        (HELVETICA 10 BRR)
```

```
        (HELVETICA 8 BRR)
        (MODERN 8 BRR))
 (LITTLEFONT 3
        (HELVETICA 8)
        (HELVETICA 6 MIR)
        (MODERN 8 MIR))
 (BIGFONT 4
        (HELVETICA 12 BRR)
        (HELVETICA 10 BRR)
        (MODERN 10 BRR))
 (USERFONT BOLDFONT)
 (COMMENTFONT LITTLEFONT)
 (LAMBDAFONT BIGFONT)
 (SYSTEMFONT)
 (CLISPFONT BOLDFONT)
 (CHANGEFONT)
 (PRETTYCOMFONT BOLDFONT)
 (FONT1 DEFAULTFONT)
 (FONT2 BOLDFONT)
 (FONT3 LITTLEFONT)
 (FONT4 BIGFONT)
 (FONT5 5
        (HELVETICA 10 BIR)
        (HELVETICA 8 BIR)
        (MODERN 8 BIR))
 (FONT6 6
        (HELVETICA 10 BRR)
        (HELVETICA 8 BRR)
        (MODERN 8 BRR))
 (FONT7 7
        (GACHA 12)
        (GACHA 12)
        (TERMINAL 12))
)
```

To change the font profile, you should edit it with DEdit. DEdit will be described in a forthcoming volume.

2.1.11.2 Setting the Font Profile

Merely editing the value of FONTPROFILE does not change the current setting of the font classes. You must execute the function **FONTPROFILE** to register the new font profile with Interlisp. It takes the form:

Function:	FONTPROFILE
# Arguments:	1
Argument:	1) PROFILE, a font profile
Value:	T.

FONTPROFILE registers the font profile given by PROFILE as the current font profile to be used by PRETTYPRINT.

2.1.12 FONT VARIABLES

Interlisp uses a variety of variables to control printing with multiple fonts. *Font configurations* are used to encapsulate the values of all relevant variables in order to avoid continual setting and resetting of individual variables.

The font variables that are used by Interlisp to control printing are described in the following paragraphs.

2.1.12.1 Font Definition Variables

FONTDEFSVARS is a list of the variables to be saved by FONTNAME. Its initial value is:

```
<-FONTDEFSVARS
(FONTCHANGEFLG FILELINELENGTH COMMENTLINELENGTH FIRSTCOL
 PRETTYLCOM LISTFILESTR FONTPROFILE FONTESCAPECHAR)
```

You may add other variable names to its value that you want incorporated into a font configuration.

2.1.12.2 Current Font Configurations

FONTDEFS is a list of the current font configurations, represented as an association list of the form:

(<name> . <parameter-pairs>)

The initial value of FONTDEFS is:

```
<-FONTDEFS
((STANDARD
    (FONTCHANGEFLG . ALL)
    (FILELINELENGTH . 102)
    (COMMENTLINELENGTH 116 . 126)
    (FIRSTCOL . 60)
    (PRETTYLCOM . 25)
    (LISTFILESTR . "                              ")
    (FONTPROFILE
      ((DEFAULTFONT 1
            (TITAN 10)
            (TITAN 10)
            (TITAN 10))
      (BOLDFONT 2
            (HELVETICA 10 BRR)
            (HELVETICA 8 BRR)
           (MODERN 8 BRR))
      (LITTLEFONT 3
            (HELVETICA 8)
            (HELVETICA 6 MIR)
            (MODERN 8 MIR))
      (BIGFONT 4
            (HELVETICA 12 BRR)
            (HELVETICA 10 BRR)
            (MODERN 10 BRR))
      (USERFONT BOLDFONT)
      (COMMENTFONT LITTLEFONT)
      (LAMBDAFONT BIGFONT)
      (SYSTEMFONT)
      (CLISPFONT BOLDFONT)
      (CHANGEFONT)
      (PRETTYCOMFONT BOLDFONT)
      (FONT1 DEFAULTFONT)
      (FONT2 BOLDFONT)
```

```
(FONT3 LITTLEFONT)
(FONT4 BIGFONT)
(FONT5 5
      (HELVETICA 10 BIR)
       (HELVETICA 8 BIR)
       (MODERN 8 BIR))
(FONT6 6
       (HELVETICA 10 BRR)
       (HELVETICA 8 BRR)
       (MODERN 8 BRR))
(FONT7 7
       (GACHA 12)
       (GACHA 12)
       (TERMINAL 12)))
and so on ...
```

The configurations that are found in the initial version of FONTDEFS are STANDARD, PARC, and SMALL.

2.1.12.3 Font Escape Character

FONTESCAPECHAR specifies the font escape character that is used to signal to PRETTYPRINT that a font escape sequence has been started. Its value must be a character or a string. Its initial value is:

```
<-FONTESCAPECHAR
^|F
```

2.12.1.4 Font Change Flag

FONTCHANGEFLG enables or disables the use of multiple fonts during printing by PRETTYPRINT. If T, printing with mulitple fonts is enabled. Its initial value is:

```
<-FONTCHANGEFLG
ALL
```

2.1.12.5 Comment Line Length

COMMENTLINELENGTH is used to inform Interlisp about font widths. When FONTCHANGEFLG has the value T, the CAR of COMMENTLINELENGTH is the line length used to print comments in the right margin and its CDR is the line length used to print those for the full width.

2.1.12.6 Fonts in Memory

The fonts which have been loaded in memory are described by the variable **\FONTSINCORE**, which takes the initial form:

```
((MODERN
   (10
      ((MEDIUM REGULAR REGULAR)
       (0 (DISPLAY . {FONTDESCRIPTOR}#62,5464)))
    ))
 (TITAN
   (8
      ((BOLD REGULAR REGULAR)
```

```
      (90 (4045XLP . {FONTDESCRIPTOR}#62,5670))
      (0  (4045XLP . {FONTDESCRIPTOR}#73,45464))
   ))
  (12
     ((BOLD REGULAR REGULAR)
      (90 (4045XLP . {FONTDESCRIPTOR}#73,45054))
      (0  (4045XLP . {FONTDESCRIPTOR}#73,45540)))
   )
  (10
     ((BOLD REGULAR REGULAR)
      (90 (4045XLP . {FONTDESCRIPTOR}#73,45000))
      (0  (4045XLP . {FONTDESCRIPTOR}#73,45614))
      )
     ((MEDIUM REGULAR REGULAR)
      (90 (4045XLP . {FONTDESCRIPTOR}#73,45130))
      (0  (4045XLP . {FONTDESCRIPTOR}#73,45670)))
   ))
  and so forth ...
```

Note that there are individual descriptions for both MEDIUM and BOLD and for the two different rotations for the font.

2.1.12.7 Making a Font Configuration

You may make a font configuration using the function **FONTNAME**:

Function:	FONTNAME
# Arguments:	1
Argument:	1) NAME, the name for the font configuration
Value:	The name of the font configuration.

FONTNAME gathers the names and values of the variables which compose a font configuration and stores them on the system variable FONTDEFS under NAME.

2.1.12.8 Installing a Font Configuration

You may install a font configuration as the current font configuration by executing **FONTSET**:

Function:	FONTSET
# Arguments:	1
Argument:	1) NAME, the name of a font configuration
Value:	The name of the previous font configuration.

FONTSET installs the font configuration identified by NAME as the current font configuration, e.g., it sets the values of the font variables to the values specified in the font configuration. Consider the following example:

```
<-(FONTSET 'STANDARD)
STANDARD
```

If NAME does not correspond to a font configuration, FONTSET returns an error as follows:

```
<-(FONTSET 'X)
X is not a defined font configuration.
```

2.2 HARDWARE FUNCTIONS

Interlisp appeared to the user as the native *operating system* for the Xerox Artificial Intelligence workstations. It incorporated a number of functions that allowed the user to control various aspects of the hardware which make up the workstation configuration.

In Medley Interlisp, these kernel functions have been (or will be?) replaced by function calls to services provided by the respective operating systems under which it will run on modern systems.

2.2.1 DISPLAY SCREEN FUNCTIONS

Interlisp includes functions that give you control over operating characteristics of the display screen.

2.2.1.1 Setting the Display Screen Color

The standard black-and-white display screen interprets a 0 bit as white and a 1 bit as black under the current convention. **VIDEOCOLOR** allows you to invert the interpretation of the 0 and 1 bits. It takes the form:

Function:	VIDEOCOLOR
# Arguments:	1
Arguments:	1) BLACKFLG, a flag specifying the interpretation of the 0 bit
Value:	The previous setting.

VIDEOCOLOR is a nospread function.

BLACKFLG determines how the 0 bit will be interpreted. If BLACKFLG is NIL, a 0 bit is displayed as white and a 1 bit will be displayed as black. Setting BLACKFLG to a non-NIL value (typically T) causes 0 bits to be displayed as black and 1 bits to be displayed as white. When this expression is executed, all bits on the screen are reversed 0 for 1 and 1 for 0 (e.g., white on black).

```
<-(VIDEOCOLOR T)
NIL
```

If BLACKFLG is not given, the previous setting of BLACKFLG is returned:

```
<-(VIDEOCOLOR)
T
```

To reset the screen to the normal black-on-white display, you should use:

```
<-(VIDEOCOLOR NIL)
T
```

2.2.1.2 Setting the Video Refresh Rate

You may set the video refresh rate using the function **VIDEORATE**:

Function:	VIDEORATE
# Arguments:	1
Arguments:	1) TYPE, the type of refresh
Value:	The old value of the refresh rate.

The display screen is usually refreshed at a rate of 60 cycles per second. However, because many workstations are now used to produce videotapes for demonstrations, the Xerox AI workstations incorporate the

feature of changing the refresh rate to 60 cycles per second which is typical of most TV systems. This permits you to videotape directly from the screen.

TYPE may be either NORMAL or TAPE. NORMAL specifies the standard refresh rate for the screen. When your workstation is delivered, it will be set to this rate. TAPE specifies that the screen should be refreshed at the TV rate (which is about 60 cycles per second).

Caution: Changing the refresh rate of the display screen may alter the appearance of data displayed on the screen and change the dimensions of the display tube.

2.2.2 KEYBOARD MANAGEMENT

Interlisp provides a set of function for managing the low-level keyboard facilities. For each key on the keyboard, there is a bit in memory which may be turned on/off by the microcode. This bit is also turned on/off as the key is pressed and released by the user. You may test the key transitions as the key moves up or down. Combinations of keys results in different Ascii character codes being inserted in the system line buffer.

2.2.2.1 Testing Key Status

Many actions within Interlisp are initiated when a key is pressed. To test whether a key is pressed down, you use the function **KEYDOWNP**:

Function:	KEYDOWNP
# Arguments:	1
Arguments:	1) KEYNAME, the name of a key
Value:	T, if the key is pressed down when the function is executed; NIL, otherwise.

KEYDOWNP tests whether a specific key has been pressed down at the time that the function is executed. Most keys are given names which are recognized by KEYDOWNP as follows

1. The alphabetic and numeric keys are identified by their respective names such as A or 5.
2. The shift keys are distinguished right and left by RSHIFT and LSHIFT.
3. The space bar is identified as SPACE.
4. The tab key is identified by TAB.

Consider the following examples (where I actually press the key):

```
<-(KEYDOWNP)
ILLEGAL ARG
NIL

<-(KEYDOWNP 'A)
NIL
```

KEYDOWNP is intended to be used from within a program to detect when certain keys have been pressed by the user. Thus, its action when invoked from the keyboard may be misleading.

```
<-(KEYDOWNP 'RSHIFT)
T
```

2.2.2.2 Changing the Effects of Key Actions

When a key is pressed, the bit associated with the key in memory is used to look up the action associated with that key in an internal action table.

You may change the actions associated with a key using the function **KEYACTION**:

Function:	KEYACTION
# Arguments:	2
Arguments:	1) KEYNAME, the name of a key
	2) ACTIONS, a dotted pair specifying the actions for the key
Value:	The previous actions for the key.

If ACTIONS is NIL, then KEYACTION returns the current value of the action table for the key. Consider the following examples:

```
<-(KEYACTION 'Y)
((121 89 LOCKSHIFT) . IGNORE)

<-(KEYACTION 'RSHIFT)
(2SHIFTDOWN . 2SHIFTUP)

<-(KEYACTION 'TAB)
(IGNORE . IGNORE)
```

A *key action* is specified as a dotted pair which has the form:

(<[csDOWN-ACTION]cs> . <[csUP-ACTION]cs>)

where the actions are interpreted as the key makes the appropriate transition. The values that the ACTION may take are given in Table 2-5.

If ACTIONS is NIL, the previous setting is returned without changing the table.

Table 2-5. Key Transition Actions

Action	Effect
NIL	Take no action on this transition. Normally, all up-transitions default to NIL, e.g., we only want to know when the key has been pressed.
(char SHIFTEDCHAR LOCKFLG)	When a transition occurs, CHAR or SHIFTEDCHAR is transmitted to the system buffer. LOCKFLAG, an optional element, may take the values LOCKSHIFT or NOLOCKSHIFT.
EVENT	Place an encoding of the current state of the mouse and the selected keys in the mouse event buffer when this transition is detected.
1SHIFTUP	Change the status of the internal shift flag for the left shift key on an up transition.
1SHIFTDOWN	Change the status of the internal shift flag for the left shift key on an up transition.
LOCKUP	Change the status of the internal shift lock flag.
CTRLUP	Changes the status of the internal control key flag on the up transition.
METAUP	Change the status of the internal meta flag.
2SHIFTUP	Change the status of the internal shift flag for the right shift key on the up transition.
2SHIFTDOWN	Change the status of the internal shift flag for the right shift key on the down transition.
LOCKDOWN	Changes the status of the internal lock key flag on the down transition.
CTRLDOWN	Changes the status of the internal control key flag on the down transition.
METADOWN	Changes the status of the internal meta key flag on the down transition.

----------Deprecated----------

On the Xerox 1186 keyboard, there is no key for the backslash character. However, you can enable the backslash character for the lower right key on the keypad using the following expression:

```
<-(KEYACTION    (PACK* 'KEYPAD (CHARACTER 92))
                                '((92 44 NOLOCKSHIFT) . IGNORE)))
((92 44 NOLOCKSHIFT) . IGNORE)
```

----------Deprecated----------

Handling Shifted Characters

The list form of an action determines what values are transmitted when a key is pressed according to the following rules:

- If either of the shift keys (e.g., RSHIFT or LSHIFT) is pressed, then SHIFTEDCHAR is transmitted when the key is pressed.
- If LOCKFLAG is LOCKSHIFT, then SHIFTEDCHAR is transmitted when the LOCK key is down.
- If neither of the shift keys is pressed, then CHAR is transmitted when the key is pressed.

Initially, the alphabetic keys specify LOCKSHIFT while the numeric keys do not. Thus, you explicitly want the user to press a shift key in order to obtain the special characters that reside over the numeric keys.

2.2.2.3 Modifying Key Actions

You may modify the actions of multiple keys at one time using the function **MODIFY.KEYACTIONS**:

Function:	MODIFY.KEYACTIONS
# Arguments:	2
Arguments:	1) KEYACTIONS, a list of key actions to be set
	2) SAVECURRENT?, a flag to return current settings
Value:	Depends on the value of SAVECURRENT?.

KEYACTIONS is a list of key actions to be set where each element takes the form

(<[csKEYNAME]cs> . <[csACTIONS]cs>)

and ACTIONS has the form described in Section 2.2.2.2 above.

MODIFY.KEYACTIONS acts as if you performed many individual calls to KEYACTION with each of the elements of the list KEYACTIONS as its argument.

SAVECURRENT? determines whether MODIFY.KEYACTIONS returns a list of the results. If non-NIL (typically T), MODIFY.KEYACTIONS returns a list of the previous settings of all keys whose actions are modified as the result of specification in KEYACTIONS. Otherwise, it returns NIL.

The basic reason for using SAVECURRENT? is to be able to change the effects of keys within your program, but to be able to reset their original actions upon exit or CTRL-D via RESETFORM.

2.2.2.4 Specifying the Meta Key

The *metashift ke*y turns on the eighth (or high order) bit of any byte when characters are transmitted by pressing a key. Metashifting is used to increase the character set codes from 128 (e.g., seven bits) to 256 (e.g., 8 bits).

Depending on the keyboard that you have, the META may or may not be labeled (on the Xerox 1186 it is in the lower left corner of the primary keypad)

You may specify that the bottom blank key will be interpreted as the metashift key using the function **METASHIFT**:

Function:	METASHIFT
# Arguments:	1
Arguments:	1) FLG, a flag indicating whether metashifting should be enabled or not.
Value:	The key action corresponding to metashifting the character.

METASHIFT is a nospread function. If FLG is non-NIL (typically T), the bottom left blank key will be interpreted as the metashift key. Consider the example:

```
<-(METASHIFT)
((5 7 NOLOCKSHIFT) . IGNORE)
```

Whenever a key is pressed while the metashift key is held down, the Ascii code associated with the key will be OR'ed with 200Q in order to set the high order bit.

2.2.2.5 Getting A Key Value

You may wait until the user actually presses a key using the function **\GETKEY**:

Function:	\GETKEY
# Arguments:	0
Arguments:	N/A
Value:	The character code of the key that was pressed.

Consider the function WAIT.ON.KEY, defined below, which waits for the user to press any one of a specified number of keys (given by KEYLST). It is defined as:

```
<-(DEFINEQ          (WAIT.ON.KEY (KEYLST)
        (if         (MEMBER (\GETKEY) KEYLST)
        T hen   T)
    ))
(WAIT.ON.KEY)
```

Now, we can try this function as follows:

```
<-(WAIT.ON.KEY '(74))
<Here, I press a "J">
T
<-(WAIT.ON.KEY '(74))
<Here, I press an "X">
NIL
```

This function is quite useful if you want to force the user to press keys in a required sequence or are coding keys to represent specific functions.

2.2.3 HARDCOPY FACILITIES

----------Deprecated----------

Interlisp provided a set of facilities for generating hardcopy output in two formats: *Interpress* and *Press*.

Interpress was the format used for communicating documents between Xerox workstations and Xerox printing systems such as the 8044 and the 5700. Press was a format used to communicate documents to Xerox laser xerographic printers such as the 2700 and the 4045.
----------Deprecated----------

The hardcopy facilities allow you to reproduce text, graphics, and image displays on hardcopy media whether it be paper, transparency foils, or film.

The hardcopy facilities have been designed to permit the easy integration of new printers with existing Xerox workstations. Interlisp generates the appropriate file format according to the type of printer to be used rather than requiring the user to be explicitly aware of the printer type.

The hardcopy functions described in the following sections allow you to produce hardcopy output on any of the printers supported by the Xerox workstations either directly or through the use of Lisp library packages.

2.2.3.1 Sending a File to a Printer

You may send a file to a printer to be printed using the function **SEND.FILE.TO.PRINTER**:

Function:	SEND.FILE.TO.PRINTER
# Arguments:	3
Arguments:	1) FILE, the name of a file
	2) HOST, the name of a host system
	3) PRINTOPTIONS, a list of options
Value:	NIL.

SEND.FILE.TO.PRINTER sends the specified file to the specified printer. The file must be located on the current host.

HOST specified the host computer system on the network, if any, which had a printer which can print that type of file. If HOST was NIL, Interlisp used the first host in the list of default printing hosts, DEFAULTPRINTINGHOST.

----------Deprecated----------
If you have a stand-alone Xerox workstation, then DEFAULTPRINTINGHOST should be set to {DSK}.
----------Deprecated----------

PRINTOPTIONS is a list of options that affects the behavior of the printer. It is organized in property list format.

----------Deprecated----------
The following options are accepted by Interpress printers:

Table 2-x. Interpress Printing Options

Option	Descriptions
DOCUMENT.NAME	The name of the document which will be printed on the title page. It must be a string. The default value is the name of the file.
DOCUMENT.CREATION.DATE	The date of the creation of the document which will appear on the header page. It must be an Interlisp integer date (IDATE format). The default value is the creation date of the file.
SENDER.NAME	The name of the user who sent the file to the printer. It will appear on the title page. It must be a string. The default value is the login name of the user

RECIPIENT.NAME	The name of the person who is receiving the document. It will appear on the title page. It must be a string. The default value is NIL, meaning no name will be listed.
MESSAGE	An additional message which may be printed on the title page. It must be a string. This might be a document classification, company name, or other identifying material such as a copyright notice.
#COPIES	The number of copies of the document to be printed. The default value is 1.
PAGES.TO.PRINT	The numbers of pages of the document to be printed represented as a list of the form.
(FIRSTPAGE# LASTPAGE#)	The default value is to print all pages of the document.
MEDIUM	The type of medium on which the document is to be printed. The default is the value of NSPRINT.DEFAULT.MEDIUM (see below).
STAPLE?	T, if the document should be stapled. Apparently, this option exists only on the larger Xerox printers.
#SIDES	The number of sides of the medium on which output should be printed. Must be an integer having the value 1 or 2 as appropriate to the printer. The default value is the value of EMPRESS#SIDES.
PRIORITY	The priority of this print request which assists the host computer in placing it in the printing queue. The value is one of HIGH, NORMAL, or LOW. The default value is the printer's default.

Press printers only recognized the following print options:

- #COPIES
- #SIDES
- DOCUMENT.CREATION.DATE

----------Deprecated----------

Consider the following exampls:

```
<-(SEND.FILE.TO.PRINTER '{DSK}<LISPFILES>STRADS>EGYPT.]
{DSK}<LISPFILES>STRADS>EGYPT.;1
```

SEND.FILE.TO.PRINTER calls the functions PRINTERTYPE and PRINTFILETYPE to determine the type of printer and the file format, respectively.

2.2.3.2 Printing Media

<<To be provided at a later date>>

A number of different types of media may be used in Xerox printers. You may explicitly use the MEDIUM print option as explained above. Alternatively, the value of the system variable NSPRINT.DEFAULT.MEDIUM will be used. It is interpreted as specified in Table 2-6.

Table 2-6. Media Types

Media Type	Description
NIL	Use the printer's default.
T	Use the first medium reported available to the printer.
Courier value	An object specifying the paper type.

A Courier object must be a Lisp object of type MEDIUM. It takes one of the following formats:
1. (PAPER (KNOWN.SIZE TYPE))
2. (PAPER (OTHER.SIZE (WIDTH LENGTH)))

The paper type must be one of:
- US.LETTER
- US.LEGAL
- A0 through A10
- ISO.B0 through ISO.B10
- JIS.B0 through JIS.B10

The IRM suggests that that if you use A4 paper exclusively (which is standard US letter size 8.5 x 11 inches), it should be sufficient to set NSPRINT.DEFAULT.MEDIUM to (PAPER (KNOWN.SIZE "A4")).
If you use paper of different sizes, you should be aware that you may have to change the value of the system variable DEFAULTPAGEREGION, which specifies the region on the page used for printing. It is measured in micas fron the lower left corner of the page.

2.2.3.3 Producing Hard Copy of a Window

You may produce a hard copy image of a window on the display screen using the function **HARDCOPYW**:

Function:	HARDCOPYW
# Arguments:	6
Arguments:	1) WINDOW/BITMAP/REGION, a handle of one of these objects
	2) FILE, the name of a file
	3) HOST, the name of a host
	4) SCALEFACTOR, a reduction factor
	5) ROTATION, a rotation factor
	6) PRINTERTYPE, the type of printer
Value:	NIL.

HARDCOPYW creates a file containing an image of a bitmap suitable for printing and sends it to a printer. The size and complexity of the bitmap may be limited by the printer. WINDOW/BITMAP/REGION specifies the source of the image that is used to create the file. It is the handle of one of:
1. A window, which may be opened or closed
2. A bitmap
3. A region, which is treated as a region of the display screen

If WINDOW/BITMAP/REGION is NIL, you will be prompted to specify a region on the screen using GETREGION.

FILE is the name of the file where the output should be placed. HOST is the name of a host which has a printer of the proper type attached to it. If HOST is NIL, the file will not be printed. Otherwise, the file is sent to HOST for printing. HOST will determine the type of file using PRINTFILETYPE (see below) and print the file if it has a suitable printer attached. If FILE is NIL, a temporary file is created to hold the image. This file is sent to HOST (if non-NIL) upon completion. The temporary file is erased after the file is printed.

With both FILE and HOST having the value NIL, the default action is to print the (temporary) file.

PRINTERTYPE specifies the type of printer to use in producing the hard copy output. PRINTERTYPE must be one of the printers which implements BITMAPSCALE and whose name is found in the list which is the value of PRINTERTYPES (see below). If PRINTERTYPE is NIL, the type of hard copy file produced is determined by the first printer on DEFAULTPRINTINGHOST which implements the function BITMAPSCALE.

Consider the following examples, is depicted in Figure 2.5.

<-(HARDCOPYW (WHICHW))
NIL

Figure 2.5 Example of HARDCOPYW

2.2.3.4 Scaling

The size of the hard copy image could be scaled by specifying a reduction factor as the value of SCALEFACTOR. If no scaling factor is specified, one will automatically be computed using the source bit map size and the characteristics of the target printer.

Note: Scaling may not be supported for all printers.

2.2.3.5 Rotating the Image

ROTATION specifies how the hard copy image will be rotated on the output medium. Most printers supported only rotations of 90 degrees which allowed the document to be printed in letter style or landscape style.

2.2.3.6 Networked Printers

If a workstation resided on a network, a file could be sent directly to a printer by specifying the device name LPT as the value of FILE. Consider the following example:

<-(HARDCOPYW (WHICHW) '{LPT})
{LPT}4045XLP;1

----------Deprecated----------
Closing a file on this device caused the file to be converted to Interpress format and sent to the default printer.
----------Deprecated----------

2.2.4 DETERMINING THE PRINTER STATUS

The current status of a specified printer could be determined using the function **PRINTERSTATUS**:

Function:	PRINTERSTATUS
# Arguments:	1
Arguments:	1) PRINTER, the name of a printer
Value:	A list describing the current status.

PRINTERSTATUS returned a list, whose format depended on the type of printer, which describeed the current status of the printer. If the printer was busy and could respond in a reasonable amount of time, PRINTERSTATUS returned T.

Note: Some printers do not implement the printer status service and will not be able to respond to this query. The value of the function will be T. For example,

```
<-(PRINTERSTATUS '4045XLP)
T
```

since the 4045 laser printer was typically attached to the RS232 port of the 1186.

2.2.5 DETERMINING THE FILE FORMAT

The printing format of the file could be determined using the function **PRINTFILETYPE**:

Function:	PRINTFILETYPE
# Arguments:	1
Arguments:	1) FILE, the name of a file
Value:	The file format.

PRINTFILETYPE determined the printing file format of its argument. It returned a value which could be one of TEDIT, <<other types to be provided>>, - e.g. one of the values of PRINTFILETYPES. Consider the following examples:

```
<-(CNDIR '{DSK}<LISPFILES>STRADS>)
{DSK}<LISPFILES>STRADS>
<-(PRINTFILETYPE 'AC2)
TEXT
```

If it cannot determine the printing file format, it returned NIL.

2.2.6 DETERMINING THE PRINTER TYPE

The printer type of the printer associated with HOST could be determined using the function **PRINTERTYPE**:

Function:	PRINTERTYPE
# Arguments:	1
Arguments:	1) HOST, the name of a host
Value:	The type of the printer; otherwise, NIL.

PRINTERTYPE uses the following rules to determine the type of printer:

- If the value of HOST is a list, its structure should be (<printer type> <printer name>);
- If HOST is a literal atom with a non-NIL PRINTERTYPE property, the value of the property is returned as the printer type;
- If HOST contains a colon, it is assumed to be an Interpress printer;
- If HOST is the CADR of a list on DEFAULTPRINTINGHOST, the CAR of that list if
- the type of printer;
- Otherwise, the value of the global system variable DEFAULTPRINTERTYPE is returned as the value.

2.2.7 HARDCOPY SYSTEM VARIABLES

A set of system variables was used to control the behavior of the hardcopy functions and to describe the printer environment available to Interlisp systems.

2.2.7.1 Printer Types

The system variable PRINTERTYPES is a list of printers recognized by Interlisp. Each element of the list is an expression of the form:

```
(<type>
   (<property-1> <value-1>)
      . . . . . .
   (<property-N> <value-N>))
```

The <type> field is a list of the printer types defined by this entry. The property list entries define properties associated with each printer type.

The properties which may be specified for printers are described in Table 2-7.

Table 2-7. Printer Properties

Property	Description
CANPRINT	Its value is a list of the file types that the printer can print directly.
STATUS	Its value is a function that is able to determine and return the status of the printer in response to an invocation of
PRINTERSTATUS	(Section 2.2).
PROPERTIES	Its value is a function which returns a list of known printer properties.
SEND	Its value is a function which invokes the appropriate protocol to send to the printer.
BITMAPSCALE	Its value is a function of the arguments WIDTH and HEIGHT (in bits) which returns a scale factor to be used in scaling the bitmap.
BITMAPFILE	Its value is an expression which converts a bitmap to a file format that the printer will accept (e.g., print).

<<The types of printers supported in the near future.>>

----------Deprecated----------
The initial value of PRINTERTYPES is (after 4045XLPSTREAM.DCOM has been loaded):

```
(((4045XLP)
   (CANPRINT (4045XLP))
   (STATUS TRUE)
   (SEND 4045XLPPRINT)
   (HOSTNAMEP 4045XLP.HOSTNAMEP)
   (BITMAPSCALE 404XLP.BITMAPSCALE)
   (BITMAPFILE (4045XLPBITMAPFILE FILE BITMAP SCALEFACTOR REGION
     ROTATION TITLE)))
 ((INTERPRESS 8044)
   (CANPRINT (INTERPRESS))
   (HOSTNAMEP NSPRINTER.HOSTNAMEP)
   (STATUS NSPRINTER.STATUS)
   (PROPERTIES NSPRINTER.PROPERTIES)
   (SEND NSPRINT)
   (BITMAPSCALE INTERPRESS.BITMAPSCALE)
   (BITMAPFILE (INTERPRESSBITMAP FILE BITMAP SCALEFACTOR REGION
     ROTATION TITLE)))
 ((PRESS SPRUCE PENGUIN DOVER)
   (CANPRINT (PRESS))
   (STATUS PUP.PRINTER.STATUS)
   (PROPERTIES PUP.PRINTER.PROPERTIES)
```

```
    (SEND EFTP)
    (BITMAPSCALE NIL)
    (BITMAPFILE (PRESSBITMAP FILE BITMAP SCALEFACTOR REGION ROTATION
      TITLE)))
 ((FULLPRESS RAVEN)
    (CANPRINT (PRESS))
    (STATUS PUP.PRINTER.STATUS)
    (PROPERTIES NILL)
    (SEND EFTP)
    (BITMAPSCALE PRESS.BITMAPSCALE)
    (BITMAPFILE (PRESSBITMAP FILE BITMAP SCALEFACTOR REGION ROTATION
      TITLE)))
    and so on...
```

Xerox has developed and used a number of different types of printers over the past decade. Some of these remain experimental printers while others have eventually emerged as commercial products. You may see the following names used in place of certain printer types:

8044>>INTERPRESS
SPRUCE>>PRESS
PENGUIN>>PRESS
DOVER>>PRESS
RAVEN>>PRESS
----------Deprecated----------

2.2.7.2 Print File Types

The system variable **PRINTFILETYPES** is a list that contains information about the various file formats which can be printed. Each element of the list takes the form:

```
(<type>
   (<property-1> <value-1>)
      . . . . . .
   (<property-N> <value-N>))
```

The properties that may be specifiedare presented in Table 2-x.

Table 2-x. Types of Print Files

Type	Usage
TEST	Its value is a function which tests the file format to determine if it is a given type.
CONVERSION	Its value is a property list of file types and the functions which may be used to convert from the specified type to the file format.
EXTENSION	Its value is a list of possible file extensions for this file type which are used in seeking the file in a directory.

----------Deprecated----------
The initial value of PRINTFILETYPES is (after TEdit has been loaded):

```
((TEDIT     (TEST TEST.FORMATTEDP1)
            (EXTENSION (TEDIT))
 )
 (4045XLP   (TEST 4045XLPFILEP)
            (EXTENSION (4045XLP))
            (CONVERSION
```

```
                (TEXT       4045XLP.PRINTEXT
                            TEDIT
                            (LAMBDA (FILE PFILE)
                                (SETQ FILE (OPENTEXTSTREAM FILE)
                            )
                            (TEDIT.FORMAT.HARDCOPY   FILE
                                                     PFILE
                                                     T
                                                     NIL
                                                     NIL
                                                     NIL
                                                      '4045XLP
                            )
                            (CLOSEF? FILE)
                            PFILE
                )
                TEDIT
                TEDIT.HARDCOPY
            )
        )
 (INTERPRESS    (TEST INTERPRESSFILEP)
                (EXTENSION (IP RP INTERPRESS))
                (CONVERSION
                    (TEXT       MAKEINTERPRESS
                                TEDIT
                                (LAMBDA (FILE PFILE)
                                    (SETQ FILE (OPENTEXTSTREAM FILE)
                                )
                                (TEDIT.FORMAT.HARDCOPY   FILE
                                                         PFILE
                                                         T
                                                         NIL
                                                         NIL
                                                         NIL
                                                         'INTERPRESS)
                                (CLOSEF? FILE)
                                PFILE
                ))
                TEDIT
                TEDIT.HARDCOPY)
))
(PRESS  (TEST PRESSFILEP)
        (EXTENSION (PRESS))
        (CONVERSION
                (TEXT   MAKEPRESS
                        TEDIT
                        (LAMBDA (FILE PFILE)
                                (SETQ FILE (OPENTEXTSTREAM FILE)
                        (TEDIT.FORMAT.HARDCOPY   FILE
                                                 PFILE
                                                 T
                                                 NIL
                                                 NIL
                                                 NIL
                                                 'PRESS
                        )
```

```
                    (CLOSEF? FILE)
                    PFILE)
                )
))
```

----------Deprecated----------

2.2.7.3The Default Printing Host

If you do not specify a printer in a hardcopy function, Interlisp uses the value of DEFAULTPRINTINGHOST to determine where to send the file for printing. It designates the default printer to be used as the output of printing operations. The value of DEFAULTPRINTINGHOST should be a list of known printer host names that are accessible to the workstation (usually, via the Ethernet).

The value of DEFAULTPRINTINGHOST may take the form: (<printertype> <host(s)>) where <printertype> is one of the values found in PRINTERTYPES and <host(s)> is a list of hosts that have that printer attached. The type of printer determines the protocol used to send information to the printer for printing.

If DEFAULTPRINTINGHOST is a single printer name, it is treated as a list of one element.

----------Deprecated----------

2.3 FLOPPY DISK MANAGEMENT

A workstation may contain a floppy disk drive in addition to an integral hard disk drive. The floppy disk drive is used for transferring files among machines as well as for archiving files which are irregularly used. the floppy disk drive is accessed through the device name {FLOPPY}. Two types of floppy disk drives have been provided for Xerox workstations: an 8" drive on the older 1100/1108/1109 models and a 5-1/4" drive provided for the 1185/1186 models. In most cases the functions described here should work for both types of drives. However, you should consult the User's Guide for your workstation to determine if there are specific conditions or characteristics of which you should be aware.

2.3.1 OPENING A STREAM TO THE FLOPPY DISK

In order to use the floppy disk drive, you must make it known to the system. You may do so by opening a stream to the floppy disk drive via the following expression:

```
<-(SETQ      FLOPPY.STREAM
             (OPENSTREAM   '{FLOPPY}LispInstallation.Script
                           'INPUT
                           'OLD))
{STREAM}#60,47470
```

Once you have opened a stream to the floppy disk, you may utilize the standard Interlisp input/output functions to read, write, and control the floppy disk consonant with the modes in which you have opened the floppy disk.

2.3.2 SETTING THE FLOPPY DISK MODE

Interlisp supports a number of modes for reading and writing floppy disks as presented in Table 2-x.

Table 2-x. Floppy Disk Modes

Mode	Description
PILOT	PILOT is the normal floppy disk mode. It allows the normal Interlisp I/O functions to be performed on the floppy disk. It also supports a directory and file naming convention similar to that available through the hard disk management facilities (see Section 2.5).
HUGEPILOT	HUGEPILOT is used to access files which span more than one floppy disk. When opening a floppy disk stream in this mode, you must specify the value of the LENGTH attribute so that the number of floppy disks required to accommodate the file can be calculated. When output is written to floppy disks in this mode, each floppy is automatically erased and reformatted. During the I/O operation, you will be prompted to insert the next floppy disk when the operation on the previous one has been completed.
SYSOUT	SYSOUT mode is similar to HUGEPILOT, but represents one contiguous file. It is used to store sysout files that represent a dump of virtual memory. Loading sysout files is specific to the type of workstation that you have so you should consult your User's Guide.
CPM	Interlisp supports a single-density, single-sided (SDSS) CPM format for exchanging information with other computer systems. This mode is incompatible with the PILOT mode for floppy disks.

To set the floppy disk mode, you may use the FLOPPY.MODE, which takes the following format:

Function: FLOPPY.MODE
Arguments: 1
Arguments: 1) MODE, the new mode for the floppy disk
Value: The old mode of the floppy disk.

FLOPPY.MODE sets the current mode of the floppy disk. Consider the following example:

```
<-(FLOPPY.MODE 'HUGEPILOT)
PILOT
<-(COPYFILE '{FLOPPY}USSRFILE.DCOM '{DSK}<LISPFILES>STRADS>USSRFILE.DCOM)
{DSK}<LISPFILES>STRADS>USSRFILE.DCOM)
```

After you have inserted the first disk, Interlisp copies the contents of the first disk to the file on the hard disk. It then prompts you to insert the second floppy disk. Once you have done so, click the left mouse button to indicate the disk is ready. Interlisp then proceeds to read this disk.

This process continues until all floppy disks comprising the HUGEPILOT file have been read. You may then revert to the previous mode.

```
<-(FLOPPY.MODE 'PILOT)
HUGEPILOT
```

2.3.3 FORMATTING A FLOPPY DISK

You must format a floppy disk the first time you intend to use it unless it has been supplied pre-formatted with software on it (such as the disks that are distributed by Xerox). To format a floppy disk, you may use the **FLOPPY.FORMAT**:

Function: FLOPPY.FORMAT
Arguments: 3
Arguments: 1) NAME, the name to be given to the floppy disk
2) AUTOCONFIRMFLG, a flag to prompt the user to confirm erasure of the floppy
3) SLOWFLG, a flag to improve formatting performance

Value: The floppy disk name.

FLOPPY.FORMAT erases the floppy disk and (re-)initializes the track information. Note that this implies that the floppy disks to be used must be *soft-sectored*. Formatting must be performed prior to the first time a new floppy disk is to be used or when you want to re-use a floppy disk (possibly one which has been corrupted or has a different format).

You may give each floppy disk a name. This name may later be checked (using FLOPPY.NAME) do ensure that the proper disk has been inserted into the floppy drive. The name of the floppy disk must be a string which has a total length less than 106 characters. FLOPPY.FORMAT attempts to interpret the contents of a floppy disk prior to reformatting it. If the disk appears to have valid information, FLOPPY.FORMAT will ask the user to confirm erasure and reformatting. User confirmation is requested when AUTOCONFIRMFLG has the value NIL. If AUTOCONFIRMFLG is T, you will not be prompted to confirm the erasure and reformatting of the disk.

Formatting a floppy disk is an I/O intensive process. It may cause a loss of cycles which affects other elements of the workstation such as the tracking of the mouse or acceptance of keystrokes.

The formatting process could be improved by writing only the necessary information on the disk required to create an empty directory. SLOWFLG, when given the value T, causes Interlisp to write only track information and critical PILOT records on the disk. Additional directory information will be written later as files are created on the disk. Essentially, you trade a lengthy, but complete, reformatting process up-front for longer I/O operations whenever a file is written to the disk.

2.3.4 NAMING A FLOPPY DISK

A name could be assigned to a floppy disk in order to identify it using the function **FLOPPY.NAME**:

Function:	FLOPPY.NAME
# Arguments:	1
Arguments:	1) NAME, a name to be assigned
Value:	The old name of the floppy disk.

FLOPPY.NAME assigns the specified name to the floppy disk. This is useful for identifying each flopy disk when it is inserted in the drive. A program may interrogate the floppy disk name (when it has requested a floppy disk to be inserted) in order to determine if it is the correct floppy disk. If NAME is NIL, FLOPPY.NAME just returns the current name of the floppy disk residing in the drive. Consider the following example:

```
<-(FLOPPY.NAME)
"Steve's Disk"
```

2.3.5 DETERMINING THE FREE PAGES

A floppy disk was formatted in terms of a number of pages (typically 671). You may use FLOPPY.FREE.PAGES to determine the number of free pages on the current floppy disk. It takes the form:

Function:	FLOPPY.FREE.PAGES
# Arguments:	0
Arguments:	N/A
Value:	The number of free pages.

FLOPPY.FREE.PAGES returns the number of unallocated free pages on the floppy disk that is currently inserted into the disk drive. Consider the example:

```
<-(FLOPPY.FREE.PAGES)
```

200

Floppy disks that were formatted in PILOT mode stored their files as a set of contiguous pages. The IRM recommended that such floppy disks be maintained in a state that utilized less than 75% of their capacity.

2.3.6 DETERMINING READABILITY

A user could determine if a floppy disk was inserted into the disk drive by executing the function **FLOPPY.CAN.READP**:

Function:	FLOPPY.CAN.READP
# Arguments:	0
Arguments:	N/A
Value:	T or NIL.

FLOPPY.CAN.READP returns T if there is a floppy disk inserted in the disk drive. Consider the example:

```
<-(FLOPPY.CAN.READP)
T
```

where a floppy disk was inserted into the disk drive. Note that the disk drive latch must be closed and locked in order for the disk drive to be enabled.

2.3.7 DETERMINING WRITABILITY

You can determine if the floppy disk inserted in the disk drive can be written using the function **FLOPPY.CAN.WRITEP**:

Function:	FLOPPY.CAN.WRITEP
# Arguments:	0
Arguments:	N/A
Value:	T or NIL.

FLOPPY.CAN.WRITEP returned T if there is a floppy disk inserted in the disk drive which was enabled for writing. Note that the user could not write on a floppy disk if its "write-protect notch" was punched out. Consider the following example:

```
<-(FLOPPY.CAN.WRITEP)
T
```

Now, inserting a floppy disk which has been disabled for writing:

```
<-(FLOPPY.CAN.WRITEP)
NIL
```

2.3.8 WAITING FOR FLOPPY AVAILABILITY

After requesting that a user insert a floppy disk into the disk drive, the user could force a program to wait until the floppy was inserted and properly registered (e.g., the disk drive door is shut and latched) using the function **FLOPPY.WAIT.FOR.FLOPPY**:

Function:	FLOPPY.WAIT.FOR.FLOPPY
# Arguments:	1

Arguments: 1) NEWFLG, a flag indicating when to return
Value: NIL.

FLOPPY.WAIT.FOR.FLOPPY waited until a floppy disk was properly registered in the disk drive before returning if NEWFLG had the value T. Otherwise, it returned immediately after checking if a floppy disk had been inserted. In the latter case, FLOPPY.CAN.READP could be used to see if the floppy could be read. Consider the following example:

```
<-(FLOPPY.WAIT.FOR.FLOPPY)
Floppy: Type any character after inserting new floppy.
<CR>
NIL
```

2.3.9 SCAVENGING A FLOPPY DISK

The user could scavenge a floppy disk to determine if any damage had occurred to the file structures using the function **FLOPPY.SCAVENGE**:

Function: FLOPPY.SCAVENGE
Arguments: 0
Arguments: N/A
Value: T.

FLOPPY.SCAVENGE attempted to determine the status of a floppy disk file structure and repair, upon user confirmation, critical records which had been become altered. Such altered records could cause errors during file operations. In certain cases, FLOPPY.SCAVENGE might be able to retrieve accidently deleted files if they had not been overwritten. Consider the following example:

```
<-(FLOPPY.SCAVENGE)
Scavenge contents of Floppy Steve's Disk?
Yes
.20.....40.....60.....80.....
and so forth
T
```

----------Deprecated----------

3. DISPLAY MANAGEMENT

Interlisp uses an interactive display screen to enhance the user interface for development and applications programs and, thus, makes the user more productive. The display, as discussed in Section 2.1, is a bitmapped display where each individual bit is addressable by its coordinates. Data structures displayed on the screen may be accessed by means of the mouse which is represented by a cursor appearing on the screen.

To fully utilize the power of Interlisp, one must understand the characteristics of the display environment. This chapter provides an in-depth look at the structures and functions upon which the rest of the interactive programming tools are built. Some of the functions defined in this chapter are not provided in the standard Interlisp system, but were defined by me to demonstrate how data structures and their associated functions may be used in various ways.

3.1 DISPLAY SCREEN COORDINATE SYSTEM

The display screen is implemented as an X-Y coordinate system. The actual size depends upon the workstation. At the most primitive level, the display screen can be viewed as just a collection of bits which can be turned on and off. By turning on the appropriate bits, display patterns appear on the screen when the patterns are transmitted from the display memory.

3.1.1 POSITIONS

The fundamental data structure associated with the display screen is a *position* which is comprised of a pair of numbers representing an X-axis coordinate and a Y-axis coordinate. A position is implemented by a record structure:

```
(RECORD  POSITION (XCOORD . YCOORD)
        (TYPE?    (AND (LISTP DATUM)
                       (NUMBERP (CAR DATUM))
                       (NUMBERP (CADR DATUM))
))
(SYSTEM))
```

The value returned by the create command is a *dotted pair*. The position may be passed to other functions for processing.

To create the position (100, 200), it is written as an ordered pair enclosed in parentheses for notational convenience:

```
<-(SETQ aposition (create POSITION XCOORD <- 100 YCOORD <- 200))
(100 . 200)
```

Because a position is a record structure, its components may be accessed using the standard record package notation:

```
<-aposition:XCOORD
100
<-aposition:YCOORD
200
```

3.1.1.1 A Point Function

Alternatively, a more compact notation for creating positions may be used by defining the function **POINT** which accepts two arguments and creates a position. It takes the form:

Function:	POINT
# Arguments:	2
Arguments:	1) X, a number
	2) Y, a number
Value:	A position handle.

POINT takes two numbers and creates a position from them. Because the coordinate system is based on an integer representation, POINT coerces the numbers to integers. POINT may be defined as follows:

```
<-(DEFINEQ      (point (x y)
    (create     POSITION
                XCOORD <- (AND
                          (OR     (NUMBERP X)
                                  (ERROR "ARG NOT A NUMBER" X T))
                           (FIXP X))
                YCOORD <- (AND
                          (OR     (NUMBERP Y)
                                  (ERROR "ARG NOT A NUMBER" Y T))
                           (FIXP Y))
                )
    ))
(POINT)
```

FIXP is applied to the arguments to coerce them to integer values because some of the Interlisp functions exhibit strange behavior when presented with floating point values as components of positions. Consider the following examples:

```
<-(POINT 200 200)
(200 . 200)

<-(POINT 307.45 75.98)
(307 . 75)

<-(POINT 'A 10)
ARG NOT A NUMBER
A
```

Note that I defined considerable checking for errors in this function. This is necessary in any function to assure error-free operation. However, because this function is likely to be used quite frequently, you may want to remove it.

3.1.2 TESTING A POSITION

Given an arbitrary Lisp object, you may determine if it is a position using **POSITIONP**:

Function:	POSITIONP
# Arguments:	1
Arguments:	1) X, an arbitrary lisp object
Value:	X, if it is a position.

POSITIONP returns the value of X if X is a position; otherwise, it returns NIL. Consider the following examples:

```
<-(SETQ aposition (POINT 100 200))
(100 . 200)

<-(POSITIONP aposition)
(100 . 200)

<-(SETQ bposition '(10 20))
(10 20)

<-(POSITIONP bposition)
NIL
```

but,

```
<-(SETQ bposition (CONS 10 20))
(10 . 20)

<-(POSITIONP bposition)
(10 . 20)
```

3.1.3 COMPARING POSITIONS

Given two positions, you may want to compare them to determine some ordering between the two. Typical comparisons are performed with respect to the absolute values of their respective coordinates.

Note: These functions are not included in the standard Interlisp sysout.

3.1.3.1 Equality of Positions

You may determine the equality of two positions, labeled P1 and P2, in two ways. First, they may be the same position whence EQ and EQUAL will compare the addresses of the positions and return T or NIL as appropriate. Second, if the positions P1 and P2 are different, the positions may still be equal according to their respective coordinates. Let us define a function **=POSITION** which compares two positions for equality. It takes the form:

Function:	=POSITION
# Arguments:	2
Arguments:	1) P1, a position
	2) P2, a position
Value:	T, if the positions are equal; NIL, otherwise.

The definition for =POSITION is:

```
<-(DEFINE
    (=position (p1 p2)
            (COND
                ((EQUAL p1 p2) T)
            (T
                (AND (EQUAL p1:XCOORD p2:XCOORD)
                            (EQUAL p1:YCOORD p2:YCOORD)
                ))
      )))
```

(=POSITION)

This definition incorporates a test for equality of positions as a time saving mechanism. If the positions are not equal, =POSITION compares the X-coordinates and Y-coordinates, respectively. Consider the following example:

```
<-(SETQ  aposition  (create POSITION XCOORD <- 100 YCOORD <- 200))
(100 . 200)

<-(SETQ bposition  (create POSITION XCOORD <- 100 YCOORD <- 200))
(100 . 200)

<-(=POSITION aposition bposition)
T
```

3.1.3.2 Less Than Comparison

You may determine that a position, labeled P1, is less than a position, labeled P2, by comparing the values of the coordinates. Visually, a position P1 less than a position P2 will appear below and to the left on the display screen. Let us define a function <POSITION which compares two positions such that one is less than the other. It takes the form:

Function:	<POSITION
# Arguments:	2
Arguments:	1) P1, a position
	2) P2, a position
Value:	T, if P1 is less than P2; NIL, otherwise.

The definition for <POSITION is:

```
<-(DEFINEQ
    (<position (p1 p2)
      (COND
            ((EQUAL p1 p2) NIL)
            (T
                  (AND
                        (ILESSP p1:XCOORD p2:XCOORD)
                        (ILESSP p1:YCOORD p2:YCOORD)
            ))
      )))
(<POSITION)
```

This definition incorporates a test for equality of positions as a time saving mechanism. If the positions are equal, we return NIL. If the positions are not equal, <POSITION compares the X-coordinates and Y-coordinates, respectively. Consider the following examples:

```
<-(SETQ aposition  (create POSITION XCOORD <- 80 YCOORD <- 90))
(80 . 90)

<-(SETQ bposition  (create POSITION XCOORD <- 100 YCOORD <- 200))
(100 . 200)

<-(<POSITION aposition bposition)
T
```

3.1.3.3 Greater Than Comparison

You may determine that a position, labeled P1, is greater than a position, labeled P2, by comparing the values of the coordinates. Visually, a position P1 greater than a position P2 will appear above and to the right on the display screen. Let us define a function **>POSITION** which compares two positions such that one is greater than the other. It takes the form:

Function:	>POSITION
# Arguments:	2
Arguments:	1) P1, a position
	2) P2, a position
Value:	T, if P1 is greater than P2; NIL, otherwise.

The definition for >POSITION is:

```
<-(DEFINEQ      (>position (p1 p2)
    (COND
        ((EQUAL p1 p2) NIL)
        (T
            (AND    (IGREATERP p1:XCOORD p2:XCOORD)
                    (IGREATERPLESSP p1:YCOORD p2:YCOORD)))
    ))
(>POSITION)
```

This definition incorporates a test for equality of positions as a time saving mechanism. If the positions are equal, we return NIL. If the positions are not equal, >POSITION compares the X-coordinates and Y-coordinates, respectively. Consider the following examples:

```
<-(SETQ aposition (create POSITION XCOORD <- 80 YCOORD <- 90))
(80 . 90)

<-(SETQ bposition (create POSITION XCOORD <- 100 YCOORD <- 200))
(100 . 200)

<-(>POSITION aposition bposition)
NIL
```

3.1.3.4 Other Comparisons

You may construct functions for other comparisons using these functions as guides. Some of the functions you might consider are x-coordinate only greater, lesser, or equal, and similarly for the y-coordinate.

3.1.4 CALCULATIONS ON POSITIONS

In many applications, you will find that you need to carry out arithmetic calculations upon a pair of positions or between a position and a number. The result of such calculations should be a new position. This section describes some simple functions for performing arithmetic calculations upon positions.

3.1.4.1 Scaling

To scale a position, you multiply its coordinates by a scaling factor. Let us define **SCALE** to scale a position. It takes the form:

Function:	SCALE
# Arguments:	2
Arguments:	1) P1, a position
	2) FACTOR, a scaling factor
Value:	A new position.

SCALE multiplies the coordinates of the position P1 by the scaling factor, which may be an integer or a floating point number. Because coordinates of a position are integers, the resulting values must be converted to integers before creating the new position. SCALE may be defined as follows:

```
<-(DEFINEQ      (scale (p1 factor)
    (create     POSITION
                XCOORD <- (FIXP (TIMES p1:XCOORD factor))
                YCOORD <- (FIXP (TIMES p1:YCOORD factor)))
  ))
(SCALE)
```

Consider the following examples:

```
<-(SETQ aposition  (create POSITION XCOORD <- 100 YCOORD <- 200))
(100 . 200)

<-(SETQ bposition (SCALE aposition 3))
(300 . 600)
```

Alternatively, we can define SCALE as follows:

```
<-(DEFINEQ      (SCALE (p1 factor)
          (point (ITIMES p1:XCOORD factor)
                 (ITIMES p1:YCOORD factor))
  )
(SCALE)
```

3.1.4.2 Translating a Position

You may translate a position by a fixed distance in the X and Y using the function **TRANSLATE**:

Function:	TRANSLATE
# Arguments:	2
Arguments:	1) P1, a position
	2) DELTA, a translation factor
Value:	A new position.

TRANSLATE added the translation factor, DELTA, to the X and Y coordinates of the position. DELTA may be an integer or a floating point number. TRANSLATE may be defined as follows:

```
<-(DEFINEQ      (translate (p1 delta)
    (create     POSITION
                XCOORD <- (FIXP (PLUS p1:XCOORD delta))
                YCOORD <- (FIXP (PLUS p1:YCOORD delta)))
    ))
(TRANSLATE)
```

Consider the following examples:

```
<-(SETQ aposition (create POSITION XCOORD <- 100 YCOORD <- 200))
(100 . 200)

<-(SETQ bposition (TRANSLATE aposition 375))
(475 . 575)
```

Note that DELTA may also be a negative number.

3.1.4.3 Adding Two Positions

You may add two positions by adding their respective X and Y coordinates. Let us define **+POSITION**:

Function:	+POSITION
# Arguments:	2
Arguments:	1) P1, a position
	2) P2, a position
Value:	A new position.

We may define +POSITION as follows:

```
<-(DEFINEQ      (+position (p1 p2)
      (create   POSITION
                XCOORD <- (IPLUS p1:XCOORD p2:XCOORD)
                YCOORD <- (IPLUS p1:YCOORD p2:YCOORD))
      ))
(+POSITION)
```

Consider the following examples:

```
<-(SETQ apoint (POINT 100 100))
(100 . 100)

<-(SETQ bpoint (POINT 300 . 500))
(300 . 500)

<-(+POSITION apoint bpoint)
(400 . 600)
```

3.1.4.4 Subtracting Two Positions

The function -POSITION may be defined in a manner similar to that for +POSITION. A problem arises when subtraction of a respective pair of coordinates yields a negative number. You may allow the negative number to be the true value of the coordinate. Note that if you attempt to display this position, the display system software performs a modulo function to scale it to the display systems coordinates. Alternately, you may set negative numbers to zero.

3.1.4.5 Transposition

A new position may be generated which is the transposition of the argument position using **TRANSPOSE**:

Function:	TRANSPOSE
# Arguments:	1
Arguments:	1) P1, a position

Value: A new position.

TRANSPOSE reverses the coordinates of its argument. Transposition is often used in graphics functions. We may define TRANSPOSE as follows:

```
<-(DEFINEQ      (transpose (p1)
    (create     POSITION
            XCOORD <- p1:YCOORD
            YCOORD <- p1:XCOORD)
    ))
(TRANSPOSE)
```

Consider the following example:

```
<-(SETQ bpoint (POINT 300 . 500))
(300 . 500)

<-(TRANSPOSE bpoint)
(500 . 300)
```

3.1.5 POSITION FUNCTIONS

In many applications, a number of functions may be applied to two positions such as the distance between two points.

3.1.5.1 Calculating the Distance Between Two Positions

The distance between two positions may be calculated using the function **DISTANCE**:

Function:	DISTANCE
# Arguments:	2
Arguments:	1) P1, a position
	2) P2, a position
Value:	An integer.

DISTANCE calculates the square root of the dot product of the two points. It may be defined as:

```
<-(DEFINEQ      (distance (p1 p2)
    (SQRT       (dot.product p1 p2))
    ))
(DISTANCE)
```

where DOT.PRODUCT may be defined as:

```
<-(DEFINEQ      (dot.product (p1 p2)
    (PLUS       (TIMES p1:XCOORD p2:XCOORD)
                (TIMES p1:YCOORD p2:YCOORD))
    ))
(DOT.PRODUCT)
```

Consider the following examples:

```
<-(SETQ aposition (POINT 80 90))
(80 . 90)
```

```
<-(SETQ bposition (POINT 100 200))
(100 . 200)

<-(DISTANCE aposition bposition)
161.2452
```

3.2 REGIONS

A ***region*** is a rectangular area that defines a (possibly proper) subset of coordinates on the display screen. The coordinates of a region are referenced with respect to its lower left hand corner, which is labeled (0,0). The lower left hand corner is assigned to absolute coordinates on the display screen when locating the region in the display screen coordinate system.

Regions are characterized by the coordinates of the lower left hand corner, their width and their height. A region is implemented as a record with fields named LEFT, BOTTOM, WIDTH, and HEIGHT:

```
(RECORD   REGION
      (LEFT BOTTOM WIDTH HIEGHT)
      LEFT <- -16383
      BOTTOM <- -16383
      WIDTH <- 32767
      HEIGHT <- 32767
      (ACCESSFNS
         ((TOP
            (IPLUS        (fetch (REGION BOTTOM) OF datum)
                          (fetch (REGION HEIGHT) OF datum) -1))
          (PTOP
            (IPLUS        (fetch (REGION BOTTOM) OF datum)
                          (fetch (REGION HEIGHT) OF datum)))
          (RIGHT
            (IPLUS        (fetch (REGION LEFT) OF datum)
                          (fetch (REGION WIDTH) OF datum) -1))
          (PRIGHT
            (IPLUS        (fetch (REGION LEFT) OF datum)
                          (fetch (REGION WIDTH) OF datum)))
      ))
      (TYPE?     (AND    (EQLENGTH DATUM 4)
                         (EVERY DATUM (FUNCTION NUMBERP))
      ))
      (SYSTEM))
```

The fields of a region may be accessed using the standard record package access functions. Access functions are also defined for calculating the TOP and RIGHT coordinates of the region as well (see the display above). Note that the only test for a region that Interlisp makes is to see if its datum length is 4 elements.

3.2.1 CREATING A REGION

You may create a region by executing **CREATEREGION**:

Function:	CREATEREGION
# Arguments:	4
Arguments:	1) LEFT, the X-axis coordinate of the lower left corner
	2) BOTTOM, the Y-axis coordinate of the lower left corner
	3) WIDTH, the width of the region in pixels (or bits)

4) HEIGHT, the height of the region in pixels (or bits)

Value: A region handle.

CREATEREGION returns an instance of a REGION record which has the fields set to the values of the respective arguments. Consider the following example:

```
<-(CREATEREGION 100 200 300 500)
(100 200 300 500)

<-(CREATEREGION 10 -20 100 100)
(10 -20 100 100)
```

Interlisp will create regions with negative coordinates. However, when such regions are used to create windows, the windows will appear to begin at the edge of the screen. Portions of the window may or may not be display at all (in effect, they are displayed "off the screen").

3.2.2 TESTING POSITIONS INSIDE A REGION

A point (X,Y) may be tested to determine if it is inside a region or not using **INSIDEP:**

Function:	INSIDEP
# Arguments:	3
Arguments:	1) REGION, a region handle
	2) XorPOS, an X-axis coordinate or a position
	3) Y, a Y-axis coordinate
Value:	T, if the point is in the region; otherwise, NIL.

X may take one of two values:

1. X may be a number (e.g., an integer), whence Y must also be a number. Together, the two designate a point by its absolute coordinates on the display screen.
2. X may be a position, whence it is determined if the position is inside the region.

Consider the following examples:

```
<-aregion
(100 100 200 200)

<-(INSIDEP aregion 80 90)
NIL

<-(INSIDEP aregion 110 120)
T

<-(INSIDEP aregion 100 NIL)
ILLEGAL ARG
NIL
```

3.2.2.1 Testing for the Cursor in a Region

Sometimes, it will be useful to determine if the current location of the cursor is inside a particular region. Interlisp does not provide such a function, but we can easily define one, **CURSORINSIDEP:**

Function:	CURSORINSIDEP
# Arguments:	1
Arguments:	1) REGION, a region handle

Value: T, if the current location of the cursor is inside the specified region; otherwise, NIL.

We might define CURSORINSIDEP as follows:

```
<-(DEFINEQ (cursorinsidep (region)
            (INSIDEP region LASTMOUSEX LASTMOUSEY)
))
(CURSORINSIDEP)
```

This function is useful if you define "active" regions within a window. In order to determine if some function should be executed, you must be able to determine if the cursor is in an active region.

Consider the following example:

```
<-aregion
(100 100 200 200)

<-awindow
{WINDOW}#60,123234
```

which is the window associated with the AREGION. Now, by placing the cursor inside the window, we can test if it is in the region as follows:

```
<-(CURSORINSIDEP aregion)
T
```

Note that you cannot use CURSORPOSITION because it requires that you specify a display stream or it defaults to T, which gives erroneous results.

3.2.2.2 Testing on the Border of a Region

Sometimes, it is useful to determine if a point is on the border of a window. Interlisp does not provide such a function, but we can easily define a function, **BORDERP**:

Function: BORDERP
Arguments: 3
Arguments: 1) WINDOW, a window handle
2) X, an X-axis coordinate
3) Y, a Y-axis coordinate
Value: T, if the point (X,Y) is on the border of the region; otherwise, NIL.

We might define BORDERP as follows:

```
<-(DEFINEQ (BORDERP (window XorPOS Y)
    (PROG (X REGION INNER_REGION BORDER)
      (if     (LISTP XorPOS)
          then
              (SETQ X (fetch XCOORD of XorPOS))
              (SETQ Y (fetch YCOORD of XorPOS))
          else
              (SETQ X XorPOS)
       )
       (* Retrieve the region of the window)
       (SETQ REGION
```

```
            (if (WINDOWP window)
             then
                  (WINDOWPROP window 'REGION)
            else
                  (ERROR "ARG NOT A WINDOW" window NIL)
            ))
      (SETQ BORDER (WINDOWPROP window "BORDER"))

      (* Compute the inner region of the window by the border amount)
      (SETQ INNER-REGION
            (CREATEREGION
            (IPLUS (fetch LEFT of REGION) BORDER)
             (IPLUS (fetch BOTTOM of REGION) BORDER)
            (IDIFFERENCE (fetch WIDTH of REGION) BORDER)
            (IDIFFERENCE (fetch HEIGHT of REGION) BORDER)
            ))
      (* Now a point is on the border if it is INSIDEP the original region, but outside the inner region)
      (RETURN
         (AND
            (INSIDEP REGION X Y)
            (NOT (INSIDEP INNER-REGION X Y))
         ))
       )))
(BORDERP)
```

Consider the following examples:

```
<-W1
{WINDOW}#56,141320

<-(WINDOWPROP W1 'REGION)
(134 52 224 235)

<-(WINDOWPROP W1 'BORDER)
6

<-(SETQ LMX (LASTMOUSEX))
138

<-(SETQ LMY (LASTMOUSEY))
54

<-(BORDERP W1 LMX LMY)
T
```

3.2.2.3 Testing for a Region

An arbitrary Interlisp object may be tested to determine if it is within a region using the function **REGIONP**:

Function:	REGIONP
# Arguments:	1
Arguments:	1) X, an arbitrary Lisp object
Value:	X, if X is a region; otherwise, NIL.

Note that a region is a merely a list of four elements. Moreover, when testing for a region, Interlisp ensures that every element is a number (see the definition of a region above). Consider the following examples:

```
<-(SETQ aregion (CREATEREGION 10 10 101 101))
(10 10 101 101)

<-(REGIONP aregion)
(10 10 101 101)
```

Also:

```
<-(REGIONP (LIST 10 20 30 40))
(10 20 30 40)
```

REGIONP does not distinguish between integers and real numbers as the following shows:

```
<-(REGIONP (CREATREGION 1.0 1.0 40.0 40.0))
(1.0 1.0 40.0 40.0)
```

However, passing such a region descriptor to CREATEW will cause an error:

```
<-(CREATEW (CREATEREGION 1.0 1.0 40.0 40.0))
ILLEGAL ARG
40.0
```

3.2.3 INTERSECTION OF REGIONS

The intersection of two or more regions may be computed by executing **INTERSECTREGIONS**:

Function:	INTERSECTREGIONS
# Arguments:	1 - N
Arguments:	1-N) REGION[i], each of which is a region handle
Value:	A region handle.

INTERSECTREGIONS is a nospread function. It computes the characteristics of a region which is the minimal intersection of a number of regions. If there is no intersection of the specified regions, it returns NIL. Consider the following examples:

```
<-(INTERSECTREGIONS)
(-2147483649 -2147483649 4294967298 4294967298)
```

which represents the largest region that Interlisp can internally represent using large integers.

```
<-aregion
(100 100 200 200)

<-bregion
(200 200 400 400)

<-(INTERSECTREGIONS aregion bregion)
(200 200 100 100)

<-(INTERSECTREGIONS bregion aregion)
(200 200 100 100)
```

Create a region beginning at (300 . 300) an extending for 500 pixels in the X and Y directions.

```
<-cregion
(500 500 300 300)

<-(INTERSECTREGIONS aregion cregion)
NIL
```

because the two regions do not intersect at any common point.

3.2.4 UNION OF REGIONS

You may compute the union of a number of regions by executing **UNIONREGIONS**:

Function:	UNIONREGIONS
# Arguments:	1 - N
Arguments:	1-N) REGION[i], each of which is a region handle
Value:	A region handle.

UNIONREGIONS is a nospread function. It computes a region which is the minimal union of all of the specified regions. If no regions are specified, it returns NIL. Consider the following examples:

```
<-(UNIONREGIONS)
NIL
```

Now, using the regions specified in the previous section:

```
<-(UNIONREGIONS aregion)
(100 100 500 500)

<-(UNIONREGIONS aregion cregion)
(100 100 700 700)

<-(UNIONREGIONS aregion NIL)
NON-NUMERIC ARG
NIL
```

3.2.5 TESTING FOR INTERSECTION

You may test to Determining whether one region intersects another may be computed using **REGIONSINTERSECTP**:

Function:	REGIONSINTERSECTP
# Arguments:	2
Arguments:	1) REGION1, a region handle
	2) REGION2, a region handle
Value:	T, if the regions intersect; otherwise, NIL.

Consider the following examples using the regions specified in the previous sections:

```
<-(REGIONSINTERSECTP aregion bregion)
T

<-(REGIONSINTERSECTP aregion cregion)
NIL
```

3.2.6 TESTING FOR INCLUSION

Determining whether one region is a subregion of another region may be computed by executing **SUBREGIONP**:

Function:	SUBREGION
# Arguments:	2
Arguments:	1) LARGEREGION, a region handle
	2) SMALLREGION, a region handle
Value:	T, if SMALLREGION is a subregion of LARGEREGION; otherwise, NIL.

SMALLREGION may be a proper subregion of LARGEREGION or it may be equal to it. Consider the following examples:

```
<-dregion
(125 125 50 50)

<-(SUBREGIONP dregion aregion)
NIL

<-(SUBREGIONP aregion dregion)
T
```

3.2.7 EXTENDING A REGION

A region may be extended to include another region by redefining its lower left corner, width, and height. **EXTENDREGION** takes the form:

Function:	EXTENDREGION
# Arguments:	2
Arguments:	1) REGION, a region handle
	2) INCLUDEREGION, a region handle
Value:	The region handle of REGION with its parameters modified.

The parameters of REGION are destructively modified so that the new region includes the region specified by INCLUDEREGION. Consider the following example:

```
<-(SETQ REG1 (CREATEREGION 100 100 200 200))
(100 100 200 200)

<-(EXTENDREGION REG1 (CREATEREGION 150 150 200 200))
(100 100 350 350)

<-REG1
(100 100 350 350)

<-(SETQ REG1 (CREATEREGION 100 100 200 200))
(100 100 200 200)

<-(EXTENDREGION REG1 (CREATEREGION 125 125 50 50))
(100 100 200 200)
```

3.2.8 CONSTRAINING A REGION TO A LIMIT

A region may be forced to adhere to certain limits using the function **MAKEWITHINREGION**:

Function:	MAKEWITHINREGION

# Arguments:	2
Arguments:	1) REGION, a region to be constrained
	2) LIMITREGION, the limiting region
Value:	A new value for REGION.

MAKEWITHINREGION destructively changes the value of REGION so that its left and bottom coordinates lie within the region LIMITREGION. If the dimensions of REGION are larger than LIMITREGION, the left and bottom of REGION are made to correspond to the left and bottom of LIMITREGION. Consider the following examples:

```
<-(SETQ REG1 (CREATEREGION 100 100 200 200))
(100 100 200 200)

<-(SETQ REG2 (CREATEREGION 125 125 50 50))
(125 125 50 50)

<-(MAKEWITHINREGION REG1 REG2)
(125 125 200 200)
```

So, REG1 now overlaps REG2.

If LIMITREGION is NIL, then the value of WHOLEDISPLAY (e.g., the region associated with the display screen) is used. Consider the following example:

```
<-(MAKEWITHINREGION REG1)
(100 100 200 200)
```

3.2.9 DETERMINING IF A POINT IS IN A REGION

Determining if a point is within a region uses the function **INSIDEP**:

Function:	INSIDEP
# Arguments:	3
Arguments:	1) REGION, a region specification
	2) POSORX, a number or position
	3) Y, a number
Value:	T or NIL.

INSIDEP determines if the position is within the region. POSORX may be an integer whence Y must also be present or it may be a position whence Y may be NIL. Consider the following examples:

```
<-(SETQ REG1 (CREATEREGION 100 100 200 200))
(100 100 200 200)

<-(INSIDEP REG1 75 50)
NIL

<-(INSIDEP REG1 (POINT 150 150))
T
```

If REGION is a window handle, then the region associated with the window is used to make the determination. Consider the following example:

```
<-W1
{WINDOW}#56,141320
```

```
<-(WINDOWPROP W1 'REGION)
(134 52 224 235)

<-(INSIDEP W1 (POINT 150 150))
T
```

3.3 BITMAPS

A *bitmap* is a rectangular array of pixels (e.g., picture elements). Black-and-white bitmaps have a single plane, so each pixel has the value 0 or 1. Color bitmaps have multiple planes, so their value is typically a small integer indicating the color to be displayed. When a pixel is 0, the corresponding bit on the display screen is white, while it is black when the pixel has the value 1.

A bitmap uses a *coordinate system* similar to that for regions and windows. The lower left corner has coordinates (0,0). The extent of the bitmap is represented by specifying its width and height in bits.

Bitmaps are implemented in Interlisp by a unique datatype, BITMAP. Each bitmap is described by several fields as presented in Table 3-1.

Table 3-1. BITMAP Fields

Field	Description
BITMAPWIDTH	The width of the bitmap (bits)
BITMAPHEIGHT	The height of the bitmap (bits).
BITMAPBITSPERPIXEL	The number of bits representing the color at each pixel.
BITMAPRASTERWIDTH	The number of words required to store one row of the bit map.
BITMAPBASE	The location in memory of the bit map.
BitMapHiLoc	The address of the highest bit in the storage pool allocation used to represent the image.
BitMapLoLoc	The address of the lowest bit in the storage pool allocation used to represent the image.

The width of the screen bit map is determined by the type of monitor attached to the system. Assuming 16-bit words, the BITMAPRASTERWIDTH is 72. Some system bitmaps, such as those allocated to the basic characters are allocated from the {\UNBOXEDHUNKS} storage pool.

3.3.1 CREATING A BITMAP

You may create a new instance of a bitmap by executing **BITMAPCREATE**:

Function:	BITMAPCREATE
# Arguments:	3
	1) HEIGHT, the heighth (in pixels) of the bitmap
	2) BITSPERPIXEL, the number of bits per pixel
Value:	A bitmap handle.

BITMAPCREATE creates a new bitmap having a width of WIDTH and a height of HEIGHT. The number of colors that may be represented by the bitmap is determined by 2 ** BITSPERPIXEL, where BITSPERPIXEL may be thought of as specifying the number of planes in the bitmap. If BITSPERPIXEL is NIL, Interlisp assumes a default value of 1. Consider the following example:

```
<-(SETQ bm1 (BITMAPCREATE 100 100 1))
{BITMAP}#74,124116
```

Now, inspecting the fields of the bitmap, we find:

```
{BITMAP}#74,124116
BitMapHiLoc                    27
BitMapLoLoc                    24730
BITMAPBASE                     {}#33,60232
BITMAPRASTERWIDTH              7
BITMAPHEIGHT     100
BITMAPWIDTH                    100
BITMAPBITSPERPIXEL             1
```

The WIDTH and HEIGHT of a bitmap must be specified or an error will result:

```
<-(SETQ bm1 (BITMAPCREATE))
NIL is not a Number
```

3.3.1.1 Testing For a Bitmap

You may test whether or not an arbitrary Lisp object is a bitmap using the function **BITMAPP**:

Function:	BITMAPP
# Arguments:	1
Arguments:	1) X, an arbitrary Lisp object
Value:	X, if X is a bitmap handle; otherwise, NIL.

BITMAPP tests X to determine if it is a bitmap. If so, it returns X because X is a bitmap handle. Consider the following example:

```
<-(BITMAPP bm1)
{BITMAP}#74,124116
```

3.3.1.2 Creating a Bitmap from a Window

You may create a bitmap from the contents of a window using the function **WINDOW.BITMAP**:

Function:	WINDOW.BITMAP
# Arguments:	1
Arguments:	1) WINDOW, a window handle
Value:	A bitmap handle.

This function extracts the contents of the window, according to its region specification, and creates a bitmap from it. The bitmap is sized to the window's size. Consider the following example:

```
<-(SETQ bm (WINDOW.BITMAP LOGOW))
{BITMAP}#55,3000
```

3.3.2 GETTING BITMAP CHARACTERISTICS

You may determine a bitmap's characteristics using the functions BITMAPWIDTH, BITMAPHEIGHT, and BITSPERPIXEL.

3.3.2.1 Getting a Bitmap's Width

You may access the width of a bitmap by executing **BITMAPWIDTH**:

Function:	BITMAPWIDTH
# Arguments:	1
Argument:	1) BITMAP, a bitmap handle
Value:	The width of the bitmap in pixels.

BITMAPWIDTH returns the width of the specified bitmap in pixels. Consider the following functions on some bitmaps:

```
<-(BITMAPWIDTH bm1)
100

<-(BITMAPWIDTH (SCREENBITMAP))
1440
```

(Note: This is on my Dell Laptop. Your screen width will/may vary.)

```
<-(BITMAPWIDTH (CURSORBITMAP))
16
```

Note that the width of the screen bit map varies with the type of screen attached to your system.

3.3.2.2 Getting a Bitmap's Height

You may determine the height of a bitmap by executing **BITMAPHEIGHT**:

Function:	BITMAPHEIGHT
# Arguments:	1
Argument:	1) BITMAP, a bitmap handle
Value:	The height of the bitmap in pixels.

BITMAPHEIGHT returns the height of the specified bitmap in pixels. Consider the following examples on some bitmaps:

```
<-(BITMAPHEIGHT bm1)
100

<-(BITMAPHEIGHT (SCREENBITMAP))
900
```

(Note: This is on my Dell Laptop. Your screen width will/may vary.)

```
<-(BITMAPHEIGHT (CURSORBITMAP))
16
```

Note that the height of the screen bit map varies with the type of screen attached to your system.

3.3.2.3 Getting the Bits Per Pixel

You may determine the number of bits per pixel in a bitmap by executing **BITSPERPIXEL**:

Function:	BITSPERPIXEL
# Arguments:	1
Argument:	1) BITMAP, a bitmap handle
Value	The number of bits per pixel in the bitmap.

BITSPERPIXEL returns the number of bits used to represent each pixel. Any number greater than one indicates that color may be used. However, the physical characteristics of the display screen will mediate the actual display of the pixels. For color displays, the number of bits per pixel corresponds to the number of color planes supported by the color display.

```
<-(BITSPERPIXEL bm1)
1
```

3.3.2.4 Determining a Bitmap's Image Size

You may determine the image size of a bitmap using the function **BITMAPIMAGESIZE**, which takes the following form:

Function:	BITMAPIMAGESIZE
# Arguments:	3
Arguments:	1) BITMAP, a bitmap handle
	2) DIMENSION, which dimension
	3) STREAM, a display stream handle
Value:	The size of the bitmap image relative to the specified display stream.

Consider the following examples:

```
<-(BITMAPIMAGESIZE (CAR SAVINGCURSOR))
(16 . 16)

<-(BITMAPIMAGESIZE (CAR SAVINGCURSOR) 'WIDTH)
16

<-(BITMAPIMAGESIZE (CAR SAVINGCURSOR) 'HEIGHT)
16
```

DIMENSION may take either the value WIDTH or HEIGHT. If it is NIL, both dimensions are returned as a dotted pair as in the example above.

3.3.3 Setting Bits

You may set a bit in a bitmap by executing **BITMAPBIT**:

Function:	BITMAPBIT
# Arguments:	4
Arguments:	1) BITMAP, a bitmap handle
	2) X, an X-axis coordinate
	3) Y, a Y-axis coordinate
	4) NEWVALUE, the value of the bit
Value:	The old value of the pixel.

BITMAPBIT sets the pixel in BITMAP given by (X,Y) to NEWVALUE, if NEWVALUE is between 0 and the maximum value allowed for the bitmap. The maximum value for the pixel is determined by 2**BITSPERPIXEL. Consider the following examples:

```
<-(for I from 1 to 10
      do
      (BITMAPBIT bm1 I 10 1))
NIL
```

```
<-(for I from 1 to 10
      do
      (BITMAPBIT bm1 I 20 1))
NIL

<-(for I from 1 to 10
      do
      (BITMAPBIT bm1 10 I 1))
NIL

<-(for I from 1 to 10
      do
      (BITMAPBIT bm1 20 I 1))
NIL
```

which draws the square depicted in Figure 3.1.

[fg3.1>Example Using BITMAPBIT]fg

If NEWVALUE is NIL, the bitmap remains unaltered and the value of the pixel at (X,Y) is returned. This provides a mechanism for interrogating the bitmap to determine the current pixel value. Consider the example:

```
<-(BITMAPBIT bm1 12 12)
1
```

If NEWVALUE is negative or exceeds the maximum value for the pixel, it is assumed to be 1:

```
<-(BITMAPBIT bm1 12 12 -3)
0

<-(BITMAPBIT bm1 12 12)
1
```

If (X,Y) is a point outside the bitmap, e.g. either X or Y is negative, X exceeds the width, or Y exceeds the height, 0 is returned and no pixels are changed. Consider the following example:

```
<-(BITMAPBIT bm1 30 30)
0
```

The value of BITMAP may also be a window handle. If the window is currently open, changes to pixel values will be displayed upon the screen during the normal refresh cycle. Consider the following example:

```
<-(for I from 10 to 100
      do
      (BITMAPBIT w1 100 I 1))
NIL
```

which draws a line in the designated window.

3.3.4 COPYING A BITMAP

You may copy a bitmap by executing **BITMAPCOPY**:

Function: BITMAPCOPY

# Arguments:	1
Arguments:	1) BITMAP, a bitmap handle
Value:	A new bitmap handle.

BITMAPCOPY returns the handle of a new bitmap which is an exact duplicate of the bitmap specified as its argument. It is useful to duplicate bitmaps which are similar, but require minor editing changes. Consider the following examples:

```
<-(SETQ bbm (BITMAPCOPY abm))
{BITMAP}#61,47770
```

To get the bit map of a particular character in a font, you may do:

```
(BITMAPCOPY (GETCHARBITMAP <character> (FONTCREATE <font> <size> <face>)))
```

For example:

```
<-(SETQ x (BITMAPCOPY (GETCHARBITMAP 'J (FONTCREATE 'GACHA 10 'MRR))))
{BITMAP}#57,3250
```

3.3.5 EXPANDING AND SHRINKING A BITMAP

You may expand or shrink a bitmap in either the X or Y or both directions using the functions EXPANDBITMAP and SHRINKBITMAP.

3.3.5.1 Expanding a Bitmap

You may expand the size of a bitmap by executing **EXPANDBITMAP**:

Function:	EXPANDBITMAP
# Arguments:	3
Arguments:	1) BITMAP, a bitmap handle
	2) WIDTHFACTOR, an expansion factor for the width
	3) HEIGHTFACTOR, an expansion factor for the height
Value:	A new bitmap handle.

EXPANDBITMAP returns a new bitmap which has its width and height extended by the factors WIDTHFACTOR and HEIGHTFACTOR. Each pixel of BITMAP is copied to the new bit map and is duplicated WIDTHFACTOR times in the X direction and HEIGHTFACTOR times in the Y direction.

```
<-(SETQ cbm (BITMAPCREATE 10 10 1))
{BITMAP}#74,124212

<-(SETQ dbm (EXPANDBITMAP cbm 2 2))
{BITMAP}#61,47630

<-(BITMAPWIDTH dbm)
20

<-(BITMAPHEIGHT dbm)
20
```

If the expansion factors are NIL, the default values of 4 for WIDTHFACTOR and 1 for HEIGHTFACTOR are used. Negative values for the expansion factors for width and height are not allowed.

3.3.5.2 Shrinking a Bitmap

You may shrink a bitmap using the function **SHRINKBITMAP**:

Function:	SHRINKBITMAP
# Arguments:	4
Arguments:	1) BITMAP, a bitmap handle
	2) WIDTHFACTOR, a shrinkage factor for the width
	3) HEIGHTFACTOR, a shrinkage factor for the height
	4) DESTINATIONBITMAP, a bitmap handle
Value:	A bitmap handle.

SHRINKBITMAP returns a copy of the specified bitmap which has been shrunk in the X and y directions by the factors WIDTHFACTOR and HEIGHTFACTOR respectively. Consider the following factors:

```
<-(SETQ SMALLBM (SHRINKBITMAP BM1 2 2))
{BITMAP}#57,140226
```

The height and width of SMALLBM are 12 pixels whereas they were 25 for BM1.

If the shrinkage factors are NIL, they default to 4 for WIDTHFACTOR and 1 for HEIGHTFACTOR. If DESTINATIONBITMAP is provided, it will be used to display the revised bitmap. Otherwise, a new bitmap which has the dimensions of BITMAP reduced by 1/WIDTHFACTOR in the X-direction and 1/HEIGHTFACTOR in the Y-direction will be returned instead. WIDTHFACTOR and HEIGHTFACTOR must be positive integers.

3.3.6 READING A BITMAP

You may read in a bitmap which has been written to a file using the function **READBITMAP**:

Function:	READBITMAP
# Arguments:	1
Arguments:	1) FILE, a file name
Value:	A bitmap handle.

READBITMAP creates a bitmap by reading the specification for the bitmap which have been stored on the file by some previous execution of PRINTBITMAP. Consider the following example:

```
<-(READBITMAP 'SHK)
{BITMAP}#57,140740
```

3.3.7 PRINTING A BITMAP

A bitmap may be saved on a file (e.g., a specification of the bitmap in symbolic form) using the function **PRINTBITMAP**:

Function:	PRINTBITMAP
# Arguments:	2
Arguments:	1) BITMAP, a bitmap handle
	2) FILE, a file name
Value:	NIL.

PRINTBITMAP writes a symbolic specification of the bitmap on the file (at the current location in the file). This specification may later be read in by READBITMAP. Consider the following example:

```
<-(PRINTBITMAP (CAR SAVINGCURSOR))
(16 16
""
"FDJ"
"HJJ"
"LJJ"
"BJNL"
"JJD"
"LJD"
""
"JDN"
"KEB"
"KE"
"JMF"
"JMB"
"JEL"
""
"")
NIL
```

Interlisp writes out the bits of a bitmap as strings of characters. This ensures that the proper encoding is performed. Also, this method of storing bitmaps on files consumes less space than other methods where each pixel might be represented individually.
2

3.3.8 DISTINGUISHED BITMAPS

There are two distinguished bitmaps created by Interlisp, which are read by the workstation hardware: the SCREENBITMAP and the CURSORBITMAP.

The display screen is implemented as a bitmap whose width is given by SCREENWIDTH (usually 1024 pixels) and whose height is given by SCREENHEIGHT (usually 808 pixels). You may obtain the respective bitmap handles by executing **SCREENBITMAP** or **CURSORBITMAP**, which take the form:

FUNCTION:	SCREENBITMAP CURSORBITMAP
# Arguments:	0
Arguments:	N/A
Value:	The respective bitmap handle.

Consider the following examples:

```
<-(SCREENBITMAP)
{BITMAP}#70,167762

<-(CURSORBITMAP)
{BITMAP}#70,167770
```

3.3.9 FINDING BITS

You may determine the first bit that is set in a column or a row of a bitmap using the functions **BIT.IN.COLUMN** and **BIT.IN.ROW** respectively. They take the form:

Function:	BIT.IN.COLUMN BIT.IN.ROW
# Arguments:	2

Arguments:	1) BITMAP, a bitmap handle 2) ROW or COLUMN, an integer
Value:	The first ROW (for BIT.IN.COLUMN) or COLUMN (for BIT.IN.ROW) in which a bit is set.

These functions return the row (respectively, column) number in which a bit is set in the specified bitmap given a row or column. Consider Figure 3-1, which displays the Saving Cursor bitmap:

Figure 3-1. The Saving Cursor Bitmap

```
<-(BIT.IN.COLUMN (CAR SAVINGCURSOR) 0)
9

<-(BIT.IN.COLUMN (CAR SAVINGCURSOR) 6)
2

<-(BIT.IN.ROW (CAR SAVINGCURSOR) 2)
4
```

Note that the rows and columns are numbered, beginning with 0, from the lower left corner of the bitmap.

3.4 BITMAP MANIPULATION (BITBLT)

You may manipulate bitmaps in several ways. Typically, you will move bits from one bitmap to another. The most common example is moving bits from a scratch bitmap in memory to the bitmap representing the display screen. When bits are moved to the SCREENBITMAP, they will become visible upon the display screen.

3.4.1 MOVING BITS BETWEEN BITMAPS

You may move bits between two bitmaps using **BITBLT** (pronounced "bit-blit"):

Function:	BITBLT
# Arguments:	12
Arguments:	1) SOURCEBITMAP, a bitmap handle 2) SOURCELEFT, the X-axis coordinate of the bits in the source bitmap 3) SOURCEBOTTOM, the Y-axis coordinate of the bits in the source bitmap 4) DESTINATIONBITMAP, a bitmap handle 5) DESTINATIONLEFT, the X-axis coordinate of the bits in the destination bitmap 6) DESTINATIONBOTTOM, the Y-axis coordinate of the bits in the destination bitmap 7) WIDTH, the width of the rectangle of bits to move 8) HEIGHT, the height of the rectangle of bits to move 9) SOURCETYPE, a flag specifying how bits from the source bitmap

will be combined with those from TEXTURE
10) OPERATION, a flag specifying how bits are combined in the destination bitmap
11) TEXTURE, a texture handle
12) CLIPPINGREGION, a region specification

Value: T

BITBLT is a fairly intelligent function which seems to be rather robust. For example,

```
<-(BITBLT)
T
```

does not affect any window or bitmap. It is treated as if the bits were cast into the "bit bucket".

SOURCEBITMAP must be a bit map handle. Otherwise, strange errors tend to occur. For example,

```
<-y
{FONTDESCRIPTOR}#56,45614

<-(BITBLT y 1 1 EXECW 300 300)
FILE NOT OPEN
{FONTDESCRIPTOR}#56,45614
```

Apparently, BITBLT interprets the first argument as a file name if it is not a bitmap handle. No explanation is given in the Interlisp IRM concerning this interpretation.

SOURCELEFT and SOURCEBOTTOM specify where bits are to be taken from in the source bitmap.

DESTINATIONBITMAP is a bitmap handle of the bitmap where the bits will be copied. The location of the block (e.g., a rectangular area) where bits will be copied is specified by DESTINATIONLEFT and DESTINATIONBOTTOM. The size of the block to copy is specified by WIDTH and HEIGHT. Thus, a block of bits of size WIDTH x HEIGHT will be extracted from SOURCEBITMAP and copied to DESTINATIONBITMAP.

3.4.1.1 Source Bitmap Operations

SOURCETYPE specifies how the bits to be moved will be selected from the source bitmap. The following table summarizes the values.

Table 3-2. Source Bitmap Operations

Source Type	Usage
INPUT	Bits are selected directly from SOURCEBITMAP.
INVERT	Bits are selected from SOURCEBITMAP and inverted.
TEXTURE	Bits are selected from the TEXTURE bitmap.

The most common operation is INPUT. This is typically used in all *copy* operations. INVERT is used when one is copying bits to another bitmap and wants to highlight them. For example, if you wanted to display the changes to a bitmap, you might invert the bits. TEXTURE is principally used for filling spaces, although you may use it when copying to indicate changed areas of the bitmap.

3.4.1.2 Destination Bitmap Operations

OPERATION specifies how bits selected from the source bitmap (or TEXTURE) will be combined with bits in the destination bitmap. The following table summarizes the operations:

Table 3-3. Destination Bitmap Operations

Operation	Usage
REPLACE	Substitute source bitmap bits for destination bitmap bits.
PAINT	Perform the logical OR of the source and destination bitmap bits.
INVERT	Perform the logical XOR of the source and destination bitmap bits.
ERASE	Perform the logical AND of the source and destination bitmap bits.

REPLACE is used most frequently when copying bits from one bitmap to another. It may be considered to be a combination of ERASE and PAINT.

3.4.1.3 Clipping Regions

The *clipping region* is a region which is overlayed on an image or display stream that limits the extent in which characters will be printed and lines drawn in the stream's coordinate system. Initially, the clipping region is set to NIL which is treated as the maximum extent of the stream.

You can think of a clipping region as an invisible boundary around some subwindow of a window. When you attempt to print within the window, characters will only be printed within the region specified by the clipping region. Similarly, lines will only be drawn within the clipping region. Characters or lines which would lie outside the clipping region will not be displayed.

You should look at one or more computer graphics texts for a better exposition on clipping regions. Two books which are excellent are Newman and Sproul [newm79] and Foley and Van Dam [fole82].

3.4.2 EDITING BITMAPS

Editing a bitmap is relatively easy with the capabilities provided by **EDITBM**:

Function:	EDITBM
# Arguments:	1
Arguments:	1) BMSPEC, a bit map specification
Value:	A bitmap handle.

EDITBM is an interactive editing facility for manipulating bitmaps. It sets up the bitmap to be edited in an editing window. The editing window has two major areas: an edit area which has a grid superimposed over it and a display area in the upper left hand corner. BMSPEC is a specification for the bitmap to be edited. It can take the following values:

1. If BMSPEC is a bitmap handle, the bitmap is displayed in the Bitmap Editor window forediting.
2. If BMSPEC is an atom, its value should be a bitmap handle to be edited.
3. If BMSPEC is a region, the portion of the screen bitmap corresponding to the region is copied into the Bitmap Editor window.
4. If BMSPEC is NIL, the Bitmap Editor asks for the dimensions of the bitmap, creates a bitmap object, and displays an empty Bitmap Editor window.

Consider the following example, where BM1 is edited:

```
<-(EDITBM BM1)
{BITMAP}#57,140424
```

The Bitmap Editor window is depicted in Figure 3-2.

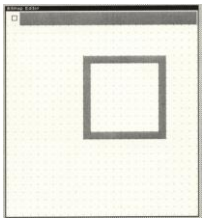

Figure 3-2. Editing a Bitmap

If the bitmap is too large to fit into the edit area, only a portion of the bitmap can be edited at a time. However, you may change the portion that you are editing by scrolling the editing pane either horizontally or vertically.

3.4.2.1 Bitmap Editor Functions

The *Bitmap Editor* provides numerous functions for editing bitmaps. These functions are accessed by pressing the mouse keys in different areas of the Bitmap Editor Window. Figure 3-3 depicts the editing functions.

The display area depicts the current bitmap as it is being created in the edit area. The entire bitmap will be displayed in the display area even though a smaller portion is being edited in the edit area. If you press the middle mouse button while the cursor is located in the display area, a menu appears that allows you to globally position the portion of the bitmap to be edited. These functions are described in Table 3-4.

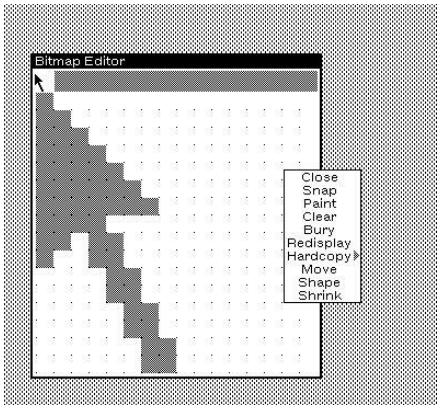

Figure 3-3. Bitmap Editing Functions

Table 3-4. Display Area Functions

Function	Description
Close	Close the bit map editor.
Snap	Move the bitmap editor to align with closest coordinates.

Paint	Paint pixels within the bitmap editor.
Clear	Clear the bitmap editor display area.
Bury	Place the bitmap editor on the oscuration stack behing the topmost window.
Redisplay	Renew the display area after editing.
Hardcopy	Print a hardcopy on a system printer of the contents of the bitmap display area.
Move	Move the bitmap editor on the screen.
Shape	Reshape the bitmap editor pane.
Shrink	Redcue the bitmap editor to an icon.

3.4.2.2 Bitmap Editing Commands

The edit area is where you make changes to the bitmap. The left mouse button causes a point to be set at the current location of the cursor (e.g., it becomes black). The middle mouse button is used to erase points at the current location of the cursor. The bitmap editing commands, described in Table 3-5, allow you manipulate the state of the Bitmap Editor.

Table 3-5. Bitmap Editing Commands

Command	Description
Paint	The Paint command copies the current bitmap into a window and invokes the window PAINT command on it (see Section 5.3.7). The PAINT command implements drawing with various brush sizes and shapes. You may exit the PAINT mode by pressing the right mouse button and selecting the QUIT command from the menu. The Bitmap Editor inquires whether or not the changes you have made via painting should be made to the bitmap itself.
Show As Tile	The ShowAsTile command tesselates the current bitmap in the display pane of the Bitmap Editor window. This allows you to determine how a bitmap will look if it were made the display screen background.
Grid On/Off	The Grid,On/Off commands turns the background grid on or off. The grid is useful for determining where to place pixels when you are constructing images, particularly new icons. It helps you to gain a sense of perspective about the image you are creating.
Grid Size	The GridSize command allows you to specify the size of the editing grid. When you select this command, another menu appears with the possible choices for the size of an editing grid.
Reset	The Reset command allows you to reset portions of the bitmap to the initial state when the Bitmap Editor was invoked. When you select Reset, a submenu is displayed which allows you to select the portion of the bitmap to be reset. When you select an item from this submenu, you are also confirming that you want some portion of the bitmap to be reset to its initial state. If you do not select any command in the submenu, no changes will be made to the bitmap.
Clear	The Clear command allows you to set portions of the bitmap to 0 (e.g., the value of WHITESHADE). As with the Reset command, a submenu is displayed which allows you to select the portion of the bitmap to be cleared.
Cursor	The Cursor<- command sets the cursor to the lower left part of the bitmap. You may then specify the "hot spot" by clicking the left mouse button after positioning the cursor in the lower left corner of the grid.
OK	The OK command allows you to record the changes you have made to the bitmap in the editing pane in the original bitmap object. It then terminates

	the Bitmap Editor and closes the Bitmap Editor window. This is the only command which allows you to record changes to the original bitmap!
Stop	The Stop terminates the Bitmap Editor without making any changes to the original bitmap. It also closes the Bitmap Editor window.

If you select an alternate grid size, the editing pane of the Bitmap Editor window is redrawn using the selected grid size. Depending on the size selected, more or less of the bitmap will be displayed in the editing pane. Scrolling is affected by a change in the grid size. The initial value of the grid size is 8. When you edit large bitmaps, it is advisable to set the grid size to a smaller value so that the edit pane encompasses more of the bitmap.

3.5 TEXTURES

A *texture* is a pattern of bits which can be used by BITBLT to create a mosaic in a bitmap. Typically, texture is used as background for windows. Textures are represented as Interlisp bitmaps and consist of 4x4 pixel patterns.

Two common textures are represented as system constants: WHITESHADE and BLACKSHADE. A global variable GRAYSHADE represents a uniform gray shade that is used by many Interlisp packages and subsystems as a background gray shade. These constants have the values:

```
<-WHITESHADE
0

<-BLACKSHADE
65536

<-GRAYSHADE
43605
```

3.5.1 CREATING TEXTURES

You may create a texture object in two ways: by extracting a 4x4 pattern from an existing bitmap or using the function EDITSHADE. To create a texture from a bitmap, you use **CREATETEXTUREFROMBITMAP**

Function:	CREATETEXTUREFROMBITMAP
# Arguments:	1
Arguments:	1) BITMAP, a bitmap
Value:	A texture object.

This function creates a texture object which represents the texture of the source bitmap. Two cases must be considered:

1. Bitmap too big
2. Bitmap too small

You must specify a bit map handle for CREATETEXTUREFROMBITMAP:

```
<-(SETQ atext (CREATETEXTUREFROMBITMAP))
ILLEGAL ARG
NIL

<-(SETQ atext (CREATETEXTUREFROMBITMAP (SCREENBITMAP)))
{BITMAP}#57,702
```

which creates a texture from the display screen bitmap. You can Display the new texture using EDITSHADE as follows:

<-(EDITSHADE atext)

which brings up the following image:

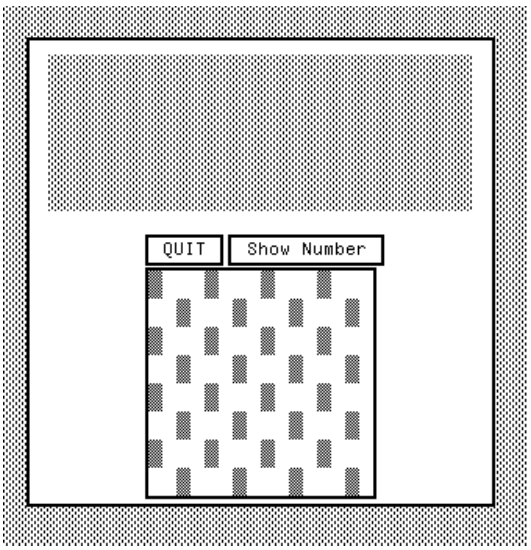

Figure 3-4. Example of a Texture

3.5.2 TESTING FOR TEXTURE

You may test whether or not an arbitrary Interlisp object is a texture object using **TEXTUREP**

Function:	TEXTUREP
# Arguments:	1
Arguments:	1) OBJECT, an arbitrary Interlisp object
Value:	OBJECT, if it is a texture; otherwise, NIL.

TEXTUREP returns OBJECT if it is a bitmap handle; otherwise, it returns NIL. Consider the following example:

<-(TEXTUREP atext)
{BITMAP}#57,702

3.5.3 INVERTING A TEXTURE

You may invert a texture, e.g., produce the analogous inverted bit map using the function **INVERT.TEXTURE**:

Function:	INVERT.TEXTURE
# Arguments:	2
Arguments:	1) TEXTURE, a texture bit map
	2) SCRATCHBM, a scratch bit map

Value: A bit map handle for the new texture.

INVERT.TEXTURE inverts the bits comprising a texture bitmap. Thus, every 0 bit becomes a 1 bit and vice versa. Consider the following example:

```
<-(SETQ atext (CREATETEXTUREFROMBITMAP (SCREENBITMAP)))
{BITMAP}#57,702

<-(SETQ btext (INVERT.TEXTURE atext))
{BITMAP}#57,3264
```

which yields the inverted texture depicted below.

```
80> (SETQ BTEXT (INVERT.TEXTURE ATEXT))
{BITMAP}#164,51424
81> (EDITSHADE BTEXT)
```

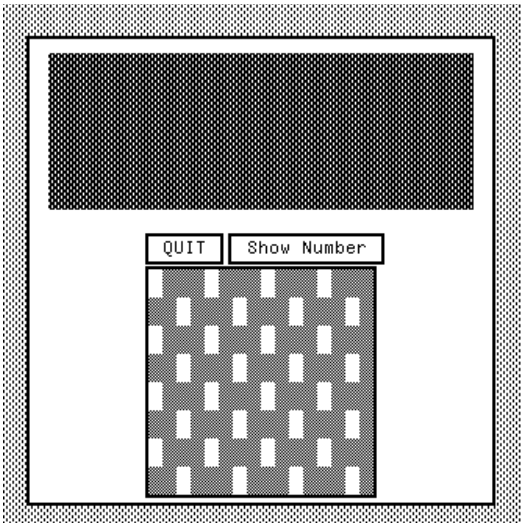

Figure 3-5. Example of Inverted Texture

If SCRATCHBM is provided, it must be a bitmap and is used to hold the resulting inverted texture.

3.5.4 EDITING A SHADE

You may edit a shade using the function **EDITSHADE**:

Function:	EDITSHADE
# Arguments:	1
Arguments:	1) SHADE, an integer specifying the shade
Value:	A bitmap handle.

EDITSHADE opens a window (actually a type of Bitmap Editor Window) for editing a texture. The texture may be either small (4x4) or large (16x16).

If SHADE is a texture object, the texture is displayed in the edit area by EDITSHADE. If SHADE is T, the edit area is initialized to a 16x16 white texture.

3.6 DISPLAY STREAMS

A *display stream* is an abstract datatype that is used as the basis for all I/O operations in Interlisp. A display stream implements localized printing of bits to specific portions of the display screen. Using a display stream, you can send characters to a window, draw lines and curves, or replace a section of a bitmap. Windows may be thought of as a special type of display stream.

3.6.1 CREATING A DISPLAY STREAM

You may create a display stream using **DSPCREATE**

Function:	DSPCREATE
# Arguments:	1
Arguments:	1) DESTINATION, a bitmap
Value:	A display stream object.

DSPCREATE creates a display stream and returns the corresponding Interlisp object. The display stream may be associated with a specific bitmap (usually a window). If DESTINATION is NIL, then the display stream is associated with the display screen bitmap. Consider the following examples:

```
<-(SETQ astream (DSPCREATE atext))
{STREAM}#62,12404

<-(SETQ bstream (DSPCREATE))
{STREAM}#62,40404
```

3.6.1.1 Display Stream Structure

A display stream is represented as an Interlisp object. The structure of a display stream is presented in Table 3-6.

Table 3-6. Display Stream Structure

Field	Datatype	Description
CHARSET	BYTE	Current character set
F10	POINTER	Device specific field
FW9	WORD	Device specific field
CBUFMAXSIZE	WORD	Maximum buffer size
STRMBOUTFN	POINTER	The BOUT function from FDEV
STRMBINFN	POINTER	The BIN function from FDEV
EXTRASTREANOP	POINTER	Used by application programs
IMAGEDATA	POINTER	Image instance variables format depends on IMAGEOPS
IMAGEOPS	POINTER	Image operations vector
OTHERPROPS	POINTER	Space for user property list
ENDOFSTREAMOP	POINTER	Used by application programs
OUTCHARFN	POINTER	Function for printing
CBUFDIRTY	BITS 5	Flag for changed bits
EOLCONVENTION	BITS 2	How EOLs are handled

LINELENGTH	WORD	Line length of stream
DIRTYBITS	WORD	Flag for changed bits
CHARPOSITION	WORD	Used by POSITION, etc.
MAXBUFFERS	WORD	Only for open streams
FW8	WORD	Only for open streams
CPAGE	WORD	Only for open streams
BUFFS	POINTER	Only for open streams
BYTESIZE	BYTE	Only for open streams
FW7	WORD	Device specific field
FW6	WORD	Device specific field
F5	POINTER	Device specific field
F4	POINTER	Device specific field
F3	POINTER	Device specific field
F2	POINTER	Device specific field
F1	POINTER	Device specific pointer
EOFFSET	WORD	EOF byte offset in buffer
EPAGE	WORD	EOF page offset in buffer
VALIDATION	POINTER	An entry determining if the file has changed
DEVICE	POINTER	FDEV entry for file
FULLFILENAME	POINTER	The name by which the file is to the user
ACCESSBITS	BITS 3	The kind of access the file is open for (read, write, append).
USERVISIBLE	FLAG	True, if listable by OPENP; NIL for terminal, dribble.
USERCLOSEABLE	FLAG	True, if stream can be closed by CLOSEF; NIL for terminal, dribble.
MULTIBUFFERHINT	FLAG	True, if the stream can read more than one buffer at a time.
REVALIDATEFLAG	FLAG	If the stream must be redisplayed.
NONDEFAULTDATEFLAG	FLAG	Device specific field
CBUFPTR	POINTER	Pointer to current buffer
EXTENDABLE	BITS 5	Permits buffer to be extended.
BOUTABLE	FLAG	Permits BOUT operations, if set.
BINABLE	FLAG	Permits BIN operations, if set.
CBUFSIZE	WORD	Offset past last byte in the buffer.
COFFSET	WORD	Offset of next BIN or BOUT operation.

You may inspect the structure of a display stream via:

<-(INSPECT my-stream)

which opens a window with the display stream properties as its contents. The following example is taken from a display stream for the terminal:

```
CHARSET              0
F10                  NIL
FW9                  0
CBUFMAXSIZE          0
STRMBOUTFN           \DSPPRINTCHAR
STRMBINFN            \STREAM.NOT.OPEN
EXTRASTREAMOP        NIL
IMAGEDATA            {\DISPLAYDATA}#70,165000
IMAGEOPS             {\IMAGEOPS}#71,47512
OTHERPROPS           NIL
```

ENDOFSTREAMOP	\EOSERROR
OUTCHARFN	\DSPPRINTCHAR
CBUFDIRTY	NIL
EOLCONVENTION	0
LINELENGTH	164
DIRTYBITS	0
CHARPOSITION	0
MAXBUFFERS	3
FW8	0
CPAGE	0
BUFFS	NIL
BYTESIZE	8
FW7	0
FW6	0
F5	NIL
F4	NIL
F3	NIL
F2	NIL
F1	NIL
EOFFSET	0
EPAGE	0
VALIDATION	NIL
DEVICE	{FDEV}#71,53524
FULLFILENAME	NIL
ACCESSBITS	6
USERVISIBLE	T
USERCLOSEABLE	NIL
MULTIBUFFERHINT	NIL
REVALIDATEFLAG	NIL
NONDEFAULTDATEFLAG	NIL
CBUFPTR	NIL
EXTENDABLE	NIL
BOUTABLE	NIL
BINABLE	NIL
CBUFSIZE	0
COFFSET	0

Many of these properties are used internally by the Interlisp kernel to manage display streams. However, a few of these may be useful to the general user. These are described below.

The LINELENGTH property specifies the maximum number of characters per line to be displayed in the stream. You can change the line length using the function LINELENGTH:

```
<-(LINELENGTH 80 bstream)
164
```

3.6.1.2 Display Data Structure

The data associated with a display stream is represented by a \DISPLAYDATA object which takes the form presented in Table 3-7.

Table 3-7. \DISPLAYDATA Structure

Field	Datatype	Description
DDSPACEWIDTH	WORD	Width (in pixels) of a space (e.g., blank)

DDCHARHEIGHTDELTA	POINTER	Internal flag for fonts
DDCHARSETDESCENT	WORD	Internal flag for fonts
DDCHARSETASCENT	WORD	Internal flag for fonts
DDCHARSET	POINTER	The current character set
DDMICARIGHTMARGIN	POINTER	
DDMICAYPOS	POINTER	
DDMICAXPOS	POINTER	
DDTexture	POINTER	The bitmap of the texture for displaying the data.
DDEOLFN	POINTER	Function for handling end-of-line condition
DDPAGEFULLFN	POINTER	Function for handling page full condition
DDCHARIMAGEWIDTHS	POINTER	Array of image widths for each character.
DDYSCALE	WORD	Y-dimension scaling factor.
DDXSCALE	WORD	X-dimension scaling factor.
DDPILOTBBT	POINTER	Pointer to the PILOTBBT data structure
XWINDOWHINT	XPOINTER	
DDHELDFLG	FLAG	
DDobsoletefield	WORD	
DDClippingTop	WORD	
DDClippingRight	WORD	
DDClippingLeft	WORD	
DDSOURCETYPE	POINTER	How bits are selected.
DDOPERATION	POINTER	Current operation.
DDScroll	POINTER	Scrolling flag
DDLeftMargin	WORD	X-coordinate for the left margin
DDRightMargin	WORD	X-coordinate for the right margin.
DDLINEFEED	WORD	
DDCOLOR	POINTER	The color in which to display the data if a color display
DDOFFSETSCACHE	POINTER	
DDWIDTHSCACHE	POINTER	Array of the distances to be moved in the X direction when each character is printed.
DDSlowPrintingCase	POINTER	
DDFONT	POINTER	Pointer to current font descriptor.
DDClippingRegion	POINTER	Clipping region specification (as a region; initially, the display stream region).
DDDestination	POINTER	Pointer to destination bitmap (usually the screen bit map).
DDYOFFSET	WORD	Current Y offset
DDXOFFSET	WORD	Current X offset
DDYPOSITION	WORD	Current Y position
DDXPOSITION	WORD	Current X position

You may inspect the structure of a \DISPLAYDATA object using the Inspector as follows:

```
<-(INSPECT '{\DISPLAYDATA}#61,53630)
```

The following example is taken from the \DISPLAYDATA object associated with the display stream in the previous section.

```
DDSPACEWIDTH          7
DDCHARHEIGHTDELTA     NIL
DDCHARSETDESCENT      0
DDCHARSETASCENT       65536
```

DDCHARSET	65536
DDMICARIGHTMARGIN	NIL
DDMICAYPOS	NIL
DDMICAXPOS	NIL
DDTexture	0
DDPAGEFULLFN	NIL
DDEOLFN	NIL
DDCHARIMAGEWIDTHS	NIL
DDYSCALE	1
DDXSCALE	1
DDPILOTBBT	{PILOTBBT}#61,147460
XWINDOWHINT	{WINDOW}#61,5240
DDHELDFLG	NIL
DDobsoletefield	0
DDClippingTop	488
DDClippingBottom	441
DDClippingRight	884
DDClippingLeft	762
DDSOURCETYPE	INPUT
DDOPERATION	REPLACE
DDScroll	NIL
DDLeftMargin	0
DDRightMargin	122
DDLINEFEED	-12
DDCOLOR	NIL
DDOFFSETSCACHE	NIL
DDWIDTHSCACHE	NIL
DDSlowPrintingCase	NIL
DDFONT	{FONTDESCRIPTOR}#70,171160
DDClippingRegion	(0 0 122 47)
DDDestination	{BITMAP}#70,167762
DDYOFFSET	441
DDXOFFSET	762
DDYPOSITION	38
DDXPOSITION	0

Note PILOT was the underlying MESA-based operating system. Xerox has provided no information on this system in the Interlisp manuals.

3.6.1.3 Initial Settings for a Display Stream

Each display stream is characterized by a set of attributes that are initialized when it is created. These attributes and their initial settings are presented in Table 3-8.:

Table 3-8. Display Stream Attributes

Attribute	Initial Value
XOffset	0
YOffset	0
ClippingRegion	A REGION object
XPosition	0
YPosition	0
Texture	WHITESHADE
Font	NIL
LeftMargin	0

RightMargin	0
SourceType	INPUT
Operation	REPLACE
LineFeed	NIL
Scroll	OFF

3.6.2 DISPLAY STREAM OPERATIONS

Interlisp provides functions for manipulating the attributes of a display stream. These functions return the old value of the attribute. A value of NIL is used to query the current value of the display stream attribute without changing it. These functions do not affect the current destination bitmap of the display stream, but do influence the effect of future operations done through the display stream.

3.6.2.1 Changing the Destination Bitmap

You may change the destination bitmap associated with the display stream using **DSPDESTINATION**:

Function:	DSPDESTINATION
# Arguments:	2
Arguments:	1) DESTINATION, the address of a bitmap
	2) DISPLAYSTREAM, a display stream handle
Value:	The current destination bitmap handle.

Each display stream has an associated bitmap to which bits are sent when something is written to the display stream. Initially, this is the screen bitmap. Consider the following example:

```
<-(SETQ my-stream (DSPCREATE))
{STREAM}#64,73404
```

The user may change the destination bitmap for MY-STREAM by executing the following expressions:

```
<-(SETQ my-bitmap (BITMAPCREATE 100 100 1))
{BITMAP}#74,124074

<-(DSPDESTINATION my-bitmap my-stream)
{BITMAP}#70,167762
```

The destination bitmap will be changed only if it is non-NIL.

The user may wish to change the destination bitmap for any number of reasons. However, a few come readily to mind:

1. Different bitmaps are associated with different information presentation roles.
2. An auxiliary bitmap may be used to construct portions of displays prior to painting them on the primary bitmap in one operation.
3. You may use other bitmaps to create overlays for the primary bitmap in which the information will be displayed.

By specifying NIL as the bitmap handle, you dissociate the display stream from any bitmap:

```
<-(DSPDESTINATION NIL my-stream)
{BITMAP}#74,124074
```

You cannot substitute a window handle for the bitmap handle:

```
<-(DSPDESTINATION awindow my-stream)
ARG NOT BITMAP
{WINDOW}#60,123324
```

The user should exercise care in executing this function because it changes the destination bitmap field of the display stream. The Window Manager maintains the display stream for windows. By changing the bitmap under a display stream without using the Window Manager, you may erase an existing display.

3.6.2.2 Changing the X and Y Origins

You may change the X and/or Y origins using the functions **DSPXOFFSET** and **DSPYOFFSET**, which take the form:

Function:	DSPXOFFSET DSPYOFFSET
# Arguments:	2
Arguments:	1) OFFSET, an integer 2) DISPLAYSTREAM, a display stream handle
Value:	The previous value of DDXOFFSET or DDYOFFSET, respectively.

These functions modify the origin of the display stream's coordinate system in the destination bitmap's coordinate system. Usually, no offset will be specified and the two coordinate systems will be identical. Display streams have their own coordinate system. This allows functions writing to a display stream to specify locations relative to its origin rather than the destination bitmap. Thus, functions need not worry about translating locations relative to the destination bitmap's coordinate system.

DSPXOFFSET returns the current X offset for the display stream, which is the X origin of the display stream's coordinate system in the destination bitmap's coordinate system. It is set to XOFFSET, if it is non-NIL.

DSPYOFFSET returns the current Y offset for the display stream, which is the Y origin of the display stream's coordinate system in the destination bitmap's coordinate system. It is set to YOFFSET, if it is non-NIL.

The X and Y offsets for a display stream are initially 0.

Typically, if you are going to write a single line to the display stream, then you will find it easier to set the position with DSPXPOSITION and DSPYPOSITION and then write the line. However, if several lines are to be written to the display stream, it is easier to change the X and Y origins and then write the lines.

A window handle may be substituted for the display stream handle. Interlisp looks up the display stream handle in the window object.

You should exercise caution when executing this function as the Window Manager maintains the X and Y offsets of a window's display stream.

3.6.2.3 Changing the Clipping Region

A *clipping region* is a region which bounds the display of characters printed and lines drawn in a display stream. Usually, a display stream will be established without a clipping region and so printing or drawing may occur anywhere within the coordinate system which corresponds to that of the destination bitmap.

When multiple functions are writing to a single display stream, some mechanism is required to prevent them from overwriting each other's information. Clipping regions are used to specify subsets of the display stream's coordinate system. Each function would set up its clipping region prior to writing to the display

stream. To specify the clipping region for a display stream, you may use the function **DSPCLIPPINGREGION**:

Function:	DSPCLIPPINGREGION
# Arguments:	2
Arguments:	1) REGION, a region identifier
	2) DISPLAYSTREAM, a display stream address
Value:	The address of the old clipping region.

If no region is specified, i.e., REGION has the value NIL, DSPCLIPPINGREGION returns the current value of the clipping region for the display stream. Consider the example:

```
<-(SETQ w1 (CREATEW (CREATEREGION 500 500 500 500) "Demo Window" 4))
{WINDOW}#64,152150
<-(DSPCLIPPINGREGION NIL w1)
(0 0 492 483)
```

These coordinates are relative to the lower left corner of the window. Because the border consumes 4 pixels per side, the width is only 492. Similarly, the title bar lies within the window region and consumes 9 pixels in addition to the 8 consumed for the top and bottom borders.

3.6.2.4 Changing the Printing Position

When you write to a display stream, the X and Y positions in the display stream are updated as new characters are printed or lines are drawn. To change the X and/or Y positions where the next printing or drawing operation will commence, you may use the functions **DSPXPOSITION** and **DSPYPOSITION**, which take the form:

Function:	DSPXPOSITION
	DSPYPOSITION
# Arguments:	2
Arguments:	1) POSITION, an integer
	2) DISPLAYSTREAM, a display stream handle
Value:	The previous value of DDXPOSITION or DDYPOSITION, respectively.

Consider the following example:

```
<-(SETQ my-stream (DSPCREATE))
{STREAM}#63,14064

<-(SETQ my-bitmap (BITMAPCREATE 150 150 1))
{BITMAP}#74,124306

<-(DSPXPOSITION NIL my-stream)
0

<-(DSPYPOSITION NIL my-stream)
0
```

The values of the coordinates are 0,0 because nothing has yet been written to the display stream. Now, by writing the following message to the display stream and obtaining the X,Y coordinate values, we see:

```
<-(PRIN1 "Hi There" my-stream)
"Hi There"
```

```
<-(DSPXPOSITION NIL my-stream)
56

<-(DSPYPOSITION NIL my-stream)
787
```

Note that the positions returned are always relative to the display stream's coordinate system rather than to the coordinate system of the destination bitmap. Now, let's change the position where we will print characters in the display stream to the location (250, 250) as follows:

```
<-(DSPXPOSITION 250 w1)
0

<-(DSPYPOSITION 250 w1)
474
```

Note that we are changing the X- and Y-coordinates of the display stream associated with the window.

```
<-(PRINT "A MESSAGE AT THE NEW DISPLAY STREAM COORDINATES" w1)
"A MESSAGE AT THE NEW DISPLAY STREAM COORDINATES"
```

would display the provided message in the display stream: Now, let us check the X and Y positions of the display stream:

```
<-(DSPXPOSITION NIL w1)
0

<-(DSPYPOSITION NIL w1)
226
```

The X-coordinate is set to 0 because PRINT forces an implicit carriage return.

3.6.2.5 Specifying Window Handles

A window handle may be substituted for the display stream handle. Interlisp looks up the display stream handle in the window object. Consider the following example:

```
<-(DSPXPOSITION 50 awindow)
0

<-(DSPYPOSITION 25 awindow)
0
```

You may specify negative values for either the X- or Y-coordinates or both. However, since they lie outside the clipping region, any print statements will not be reflected in the associated window. They do, however, modify the values of the X- and Y-coordinates.

3.6.2.6 Getting the Relative Position

You may obtain the relative screen coordinates of a position relative to the coordinates of the given display stream using the functions **DSPXSCREENTOWINDOW** and **DSPYSCREENTOWINDOW**, which take the form:

Function: DSPXSCREENTOWINDOW
DSPYSCREENTOWINDOW

# Arguments:	2
Arguments:	1) COORD, an X-coordinate
	2) STREAM, a display stream handle
Value:	The X-coordinate (or Y-coordinate) relative to STREAM.

These functions return a relative screen coordinate of the given coordinate to the origin of the display stream. Consider the following example:

```
<-(DSPXSCREENTOWINDOW 100 (TTYDISPLAYSTREAM))
-322
```

because the Interlisp Executive Window is located in the upper left hand corner of the display screen.

3.6.2.7 Changing the Texture

The texture of the background pattern of the display stream may be changed using the function **DSPTEXTURE**:

Function:	DSPTEXTURE
# Arguments:	2
Arguments:	1) TEXTURE, a texture object identifier
	2) DISPLAYSTREAM, a display stream descriptor
Value:	The previous value of DDTexture.

Initially, the texture of the display stream has the value WHITESHADE. Other standard values that may be used are GRAYSHADE and BLACKSHADE. In fact, the texture is actually an integer, so you may want to experiment with the values of various integers to see the different effects that they produce. A value of NIL returns the current value of DDTexture.Consider the following example:

```
<-(DSPTEXTURE NIL w1)
65536
```

3.6.2.8 Changing the Display Font

The font with which the next characters will be printed to the display stream may be changed using the function **DSPFONT**:

Function:	DSPFONT
# Arguments:	2
Arguments:	1) FONT, a font descriptor
	2) DISPLAYSTREAM, a display stream descriptor
Value:	The previous value of DDFont.

DSPFONT returns a font descriptor. A value of NIL returns the current value of DSPFONT. For example, in the Interlisp Executive Window:

```
<-(DSPFONT)
{FONTDESCRIPTOR}#70,171260

<-(DSPFONT NIL w1)
{FONTDESCRIPTOR}#70,171670
```

Initially, the display font for each display stream is Gacha 10. To obtain the display stream's font family, you must extract the value of the font property FAMILY from the font descriptor. Here is a little function to do that for you:

```
(DEFINEQ
    (DSP.FONT.FAMILY (stream)
        (FONTPROP (DSPFONT NIL stream) 'FAMILY)
    ))
```

So, applying this function to the Interlisp Executive Window:

```
<-(DSP.FONT.FAMILY)
HELVETICA
```

3.6.2.9 Changing the Left Margin

When writing text to a display stream, you will usually want to indent the beginning of each new line from the edge of the bitmap in which the text is displayed. This indentation is referred to as the ***left margin*** of the display stream. It is initially 0. You may change the left margin of the display stream using the function **DSPLEFTMARGIN**:

Function:	DSPLEFTMARGIN
# Arguments:	2
Arguments:	1) XPOSITION, an integer
	2) DISPLAYSTREAM, a display stream descriptor
Value:	The previous value of DDLeftMargin.

When XPOSITION is NIL, DSPLEFTMARGIN returns the current value of DDLeftMargin for the specified display stream.

```
<-(DSPLEFTMARGIN)
0
```

If DISPLAYSTREAM is NIL, the current display stream is used (e.g., that returned by TTYDISPLAYSTREAM).

3.6.2.10 Changing the Right Margin

To prevent text from butting up against the right side of the destination bitmap, you will usually want to specify a ***right margin*** for a display stream. A right margin is the maximum X position to which characters will be printed in the display stream's coordinate system. Initially, right margin is set to (SCREENWIDTH). When printing functions reach the right margin, they automatically insert an end-of-line character. You may change the right margin using **DSPRIGHTMARGIN**:

Function:	DSPRIGHTMARGIN
# Arguments:	2
Arguments:	1) XPOSITION, an integer
	2) DISPLAYSTREAM, a display stream descriptor
Value:	The previous value of DDRightMargin.

Changing the value of DDRightMargin effects both the input and output operations. The value is given as the number of pixels from the left edge of the window. A value of NIL for XPOSITION returns the current value of DDRightMargin. For the Interlisp Executive Window:

```
<-(DSPRIGHTMARGIN)
545
```

which reflects the current size of the window.

3.6.2.11 Changing the Source Type for Printing

When writing text to a display stream, the source type for bit-blitting is normally INPUT. This means that the characters will be displayed black-on-white in the window associated with the display stream. You may change the source type using **DSPSOURCETYPE**:

Function:	DSPSOURCETYPE
# Arguments:	2
Arguments:	1) SOURCETYPE, the BITBLT source type
	2) DISPLAYSTREAM, a display stream handle
Value:	The previous value of DDSourceType.

The value of SOURCETYPE may be either INPUT or INVERT. INPUT is the normal value. INVERT specifies that the bits of the characters are inverted before being displayed. A value of NIL for SOURCETYPE causes the current value of DDSOURCETYPE to be returned:

```
<-(DSPSOURCETYPE)
INPUT
```

Specifying INVERT as the source type allows you to highlight blocks of characters as they are printed on the screen. Here's a convenient function for printing with highlighting:

```
<-(DEFINEQ
    (HIGHLIGHT (msg stream)
                (DSPSOURCETYPE 'INVERT)
                (PRIN1 msg)
                 (DSPSOURCETYPE 'INPUT)
    ))
(HIGHLIGHT)
```

HIGHLIGHT takes a message and a stream as its arguments. It prints the message in highlighted form beginning at the current position of the display stream.

3.6.2.12 Changing the BITBLT Operation

When you print or draw to a display stream, the normal BITBLT operation is REPLACE, which means to overwrite bits already present at the corresponding locations in the display stream. You may change the bit-blitting operation using **DSPOPERATION**:

Function:	DSPOPERATION
# Arguments:	2
Arguments:	1) OPERATION, a BITBLT operation
	2) DISPLAYSTREAM, a display stream hand
Value:	The previous value of DDOperation.

OPERATION can be one of the BITBLT operations: PAINT, REPLACE, INVERT, or ERASE. These operations are described in Section 3.4.1.2. A value of NIL for OPERATION causes the current value of DDOperation to be returned:

```
<-(DSPOPERATION)
REPLACE
```

3.6.2.13 Changing the Linefeed Spacing

Whenever you type a line feed while entering text, the Y coordinate is incremented by the value of DDLINEFEED. Initially, the value of -12 is the negative value of the height of the initial font (Gacha 10). You may change the value of the line feed increment using the function **DSPLINEFEED**:

Function:	DSPLINEFEED
# Arguments:	2
Arguments:	1) DELTAY, the new Y coordinate increment
	2) DISPLAYSTREAM, a display stream handle
Value:	The previous value of DDLINEFEED.

If DELTAY is more negative than the current value (e.g., -15), the spacing between the lines in the display stream increases. Conversely, if it is less negative, the spacing between the lines decreases. If you wish to overwrite a line, you should set the spacing to 0. I've found that a spacing of -15 makes the text displayed in most windows very readable. A value of NIL for DELTAY returns the current value of DDLINEFEED:

```
<-(DSPLINEFEED)
-12
```

3.6.2.14 Changing the Scrolling Behavior

When you create a display stream, any scrolling behavior is initially turned off. Thus, when you print bits at the bottom of the display stream which move out of the clipping region, they will be lost. To prevent the loss of these bits, you can enable scrolling using the function **DSPSCROLL**:

Function:	DSPSCROLL
# Arguments:	2
Arguments:	1) SWITCHSETTING, the value of the scrolling switch
	2) DISPLAYSTREAM, a display stream handle
Value:	The current value of the scrolling switch.

When the scrolling switch is turned ON, the bits in the display stream's destination bitmap are moved after any linefeed that moves the current display position out of the destination bitmap. Any bits moved out of the current clipping region are lost. This function does not adjust the values of XOffset, YOffset, or ClippingRegion in the display stream handle.

If SWITCHSETTING is NIL, DSPSCROLL just returns the current setting of the scrolling switch. Consider the following example:

```
<-(DSPSCROLL NIL my-stream)
OFF
```

Otherwise, it sets the scroll flag to the value of SWITCHSETTING, if it is non-NIL:

```
<-(DSPSCROLL 'ON my-stream)
OFF
```

3.6.2.15 Resetting the Cursor in the Display Stream

You may *home* the cursor in the display stream using the function **DSPRESET**:

Function:	DSPRESET
# Arguments:	1

Arguments: 1) STREAM, a display stream handle
Value: NIL.

DSPRESET sets the X- and Y-coordinates as follows:

- The X-coordinate of STREAM is set to the left margin.
- The Y-coordinate of STREAM is set to the top of the clipping region minus the font ascent.

If STREAM is a display stream handle, the display stream's destination bitmap will be filled with its background texture. For most display streams associated with the screen, this will be equivalent to clearing the window.

3.6.2.16 Starting a New Page

You may start a new page within a display stream using the function **DSPNEWPAGE**:

Function: DSPNEWPAGE
Arguments: 1
Arguments: 1) STREAM, a display stream handle
Value: NIL.

DSPNEWPAGE begins a new page by setting the X- and Y-coordinates as follows:

- The X-coordinate is set to the left margin.
- The Y-coordinate is set to the top margin plus the linefeed.

3.6.2.17 Printing to the Center of a Display Stream

You may print an expression in the center of a region within a display stream using the function **CENTERPRINTINREGION**:

Function: CENTERPRINTINREGION
Arguments: 3
Arguments: 1) EXP, an expression
2) REGION, a region descriptor
3) STREAM, a display stream handle
Value: NIL

CENTERPRINTINREGION prints the value of EXP in the center of the given region within the display stream. Consider the following example: The result is depicted in Figure 3-6.

```
<-(CENTERPRINTINREGION   "E = M*C \^ 2" (CREATREGION 30 30 50 50) IOSTREAM)
NIL
```

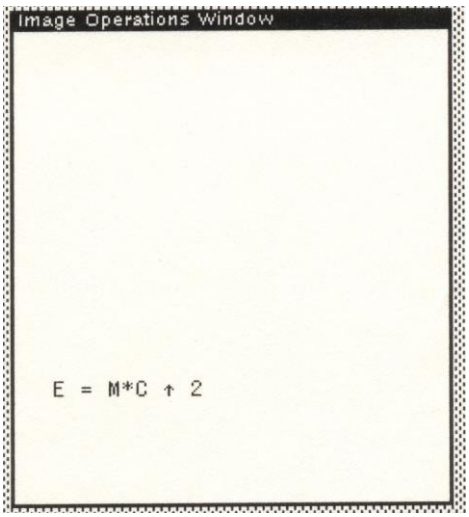

Figure 3-6. Example of CENTERPRINTINREGION

You must specify a region to be used in calculating the center.

3.6.2.18 Erasing a Character in a Display Stream

You may erase a character that has been printed in a display stream using the function **DSPBACKUP**:

Function:	DSPBACKUP
# Arguments:	2
Arguments:	1) WIDTH, the width of the character
	2) DISPLAYSTREAM, a display stream handle
Value:	T, if any bits were written; NIL, otherwise.

DSPBACKUP adjusts the current X position of the display stream by "backing it up" over a character that is assumed to be WIDTH bits wide. The area is filled with the display stream's background texture. The X position of the display stream is decreased by WIDTH bits. If this would set the X position to be less than the left margin if the display stream, the operation stops.

3.6.3 THE TTY DISPLAY STREAM

The primary file which you interact with through the Interlisp Executive window has the name T. In the display oriented environment, a standard display stream known as the TTY display stream is associated with T and is attached to the Interlisp Executive window.

When you type characters on the keyboard, the characters are sent to the window which is associated with the current TTY display stream. As you move the cursor from one window to another, Interlisp may change the TTY display stream to associate it with that window.

3.6.3.1 Changing the TTY Display Stream

You may change the TTY display stream using the function **TTYDISPLAYSTREAM**:

Function:	TTYDISPLAYSTREAM
# Arguments:	1
Arguments:	1) DISPLAYSTREAM, a display stream handle
Value:	The old display stream handle.

The value of DISPLAYSTREAM may be a window or display stream handle. The terminal output channel is switched to the new display stream. Thereafter, whenever you print characters, they will appear in the window associated with the new display stream. At the top level, one normally types into the Interlisp Executive Window. You can switch the display stream to a window called TEST via the following command (the window handle is W1):

```
<-(TYYDISPLAYSTREAM W1)
{STREAM}#71,60150
```

This action is depicted in Figure 3.9. Note that the result of executing the function is displayed in the window associated with the new display stream.

TTYDISPLAYSTREAM automatically puts the new display stream into scrolling mode. It also invokes PAGEHEIGHT with the number of lines that will fit into the display stream window given its current font and clipping region. The line length is computed from the left margin, the right margin, and the font of the display stream. Whenever one of these fields is changed, the line length is automatically recalculated. However, the page height is not automatically recalculated when one of these fields changes. You may force the page height to be recalculated using the expression:

```
<-(TTYDISPLAYSTREAM (TTYDISPLAYSTREAM))
{STREAM}#56,31000
```

The line buffer associated with the old TTY window will be saved as one of the window's properties when the TTY display stream is switched. The system line buffer becomes the one associated with the new window, if any. If none exists, one is created and associated with the window.

When you switch the TTY display stream, the associated window may not be visible because it is on the obscuration stack, it is closed, or it is shrunk. The window will not be made visible until you print or draw to the new display stream.

3.6.3.2 Changing the Caret Shape

The current input position is indicated by a *caret* which has the form indicated in Figure 3-7.

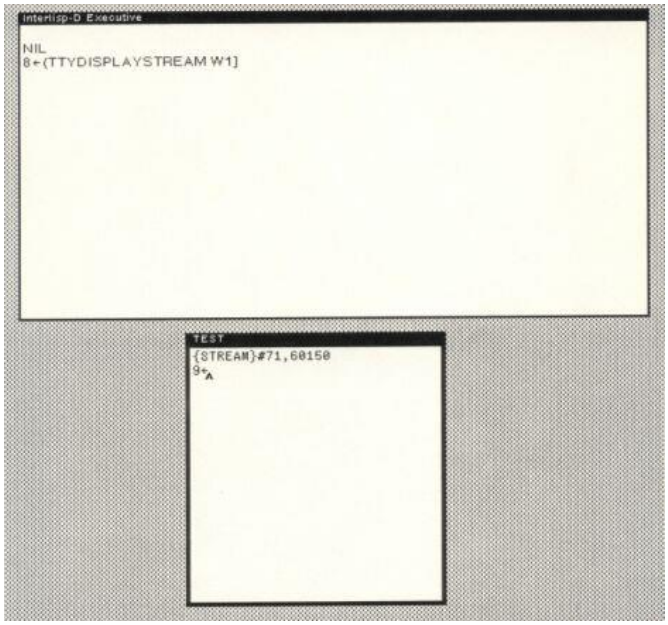

Figure 3-7. Standard Input Caret

You may change the shape of the caret to one of your own choosing using the function **CARET**:

Function: CARET
Arguments: 1
Arguments: 1) NEWCARET, a new caret specification
Value: The handle of the old caret.

CARET sets the shape of the caret which indicates the next input/output location for the TTY display stream. NEWCARET may take one of the values presented in Table 3-9.

Table 3-9. NEWCARET Values

New Caret Value	Description
NIL	No changes are made to the caret shape; the cursor handle representing the old shape is returned.
OFF	The caret is turned off.
CURSOR	A cursor handle is provided that gives the new caret shape. Its bitmap hotspot indicates which point in the new caret should be located at the current output position.

Consider the following example:

```
<-(CARET)
({BITMAP}#70,167666 3 . 4)
```

3.6.3.3 Changing the Page Height of the Display Stream

The *page height* of a display stream is computed from the left and right margins and the font size. The page height indicates the number of lines that will fit into the window displayed on the display screen which is associated with the display stream. You may change the page height using the function **PAGEHEIGHT**:

Function:	PAGEHEIGHT
# Arguments:	1
Arguments:	1) N, a number of lines
Value:	The old page height.

The current page height is stored as a property of the window with which the display stream is associated. It is automatically computed when the TTY display stream is switched to the window. You may change the page height by specifying a positive number for N. N indicates the number of lines that may be printed to the display stream before the page is held (e.g., the window contents are temporarily frozen). A page is heldt whenever the number of lines in the window reaches N+1 without intervening input. At that point, the window will be inverted until you type a character. You may set N to 0 which disables page holding and allows continuous scrolling.

3.6.4 OPENING A STRING STREAM

You may treat a string (particularly a lengthy one) as if it were a file by opening a stream to it. The stream interface allows you to read and write to the string as if it were a file. To open a stream to a string, you use the function **OPENSTRINGSTREAM**:

Function:	OPENSTRINGSTREAM
# Arguments:	2
Arguments:	1) STRING, a string
	2) ACCESS, the type of access desired
Value:	A stream handle.

OPENSTRINGSTREAM returns a stream handle which can be used in subsequent I/O operations to manipulate the characters of the string. ACCESS may be one of:

- INPUT
- OUTPUT
- BOTH

Consider the following example:

```
<-MYTEXT
"A complex polynomial is a generalization of a real number that is introduced in mathematics so that all polynomial equations with real coefficients may have solutions."

<-(SETQ MYSTREAM (OPENSTRINGSTREAM MYTEXT 'INPUT))
{STREAM}#56,34470

<-(READ MYSTREAM)
A

<-(READ MYSTREAM)
complex
```

3.7 GRIDS

A *grid* is an arbitrary partitioning of a coordinate system into rectangles. A grid is defined by its *unit grid*, which is a region which is the *origin rectangle* of the grid overlayed on the coordinate system.

3.7.1 DRAWING A GRID

To draw a grid, you must first define a *grid specification*, which is a region specifying the size of the rectangles that comprise the grid. You may draw the grid using **GRID**:

Function:	GRID
# Arguments:	6
Arguments:	1) GRIDSPEC, a region
	2) WIDTH, the width of the grid in rectangles
	3) HEIGHT, the height of the grid in rectangles
	4) BORDER, the width of the border of each rectangle of the grid
	5) STREAM, a display stream handle
	6) GRIDSHADE, the texture of the border lines
Value:	NIL.

GRID uses the grid specification given in GRIDSPEC to overlay a grid on the specified display stream. Each grid has a border that is BORDER pixels wide. Thus, the border between two grid units is 2*BORDER pixels wide. Consider the following example:

```
69> (SETQ GRIDSPEC (CREATEREGION 10 10 25 25))
(10 10 25 25)
70> (SETQ W1 (CREATEW (CREATEREGION 300 300 200 200)))
{WINDOW}#140,131664
71> (GRID GRIDSPEC 10 10 2 W1 NIL)
NIL
72>
```

which draws the grid depicted in Figure 3-8.

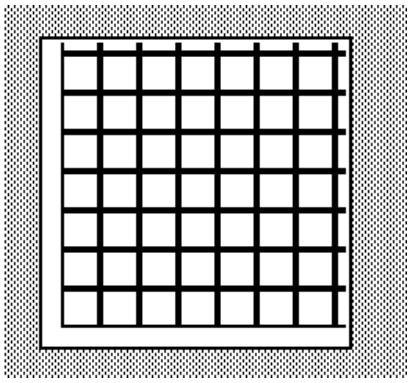

Figure 3-8. A Sample Grid

If BORDER is the atom POINT, borders will not be drawn. Rather, the lower left point of each grid rectangle will be turned on. Consider the following example:

```
<-(GRID gridspec 10 10 'POINT awindow NIL)
NIL
```

which is depicted in Figure 3-9.

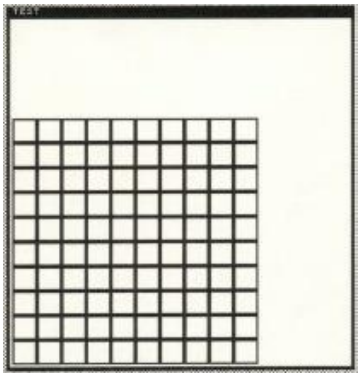

Figure 3-9. Displaying a Grid via Lower Left Corners

3.7.2 SHADING GRID UNITS

You can shade the rectangle associated with a particular grid unit using the function **SHADEGRIDBOX**:

Function:	SHADEGRIDBOX
# Arguments:	7
Arguments:	1) X, a grid rectangle identifier
	2) Y, a grid rectangle identifier
	3) SHADE, a texture handle
	4) OPERATION, ??
	5) GRIDSPEC, a grid specification
	6) BORDER, a border width
	7) STREAM, a display stream handle
Value:	NIL.

Commented [SK1]:

SHADEGRIDBOX "colors" the box specified by the grid coordinates X,Y with SHADE using the operation OPERATION. The result of shading box (5,5) is depicted in Figure 3-10.

```
75> (SHADEGRIDBOX 5 5 GRAYSHADE 'PAINT GRIDSPEC 2 W1)
NIL
76>
```

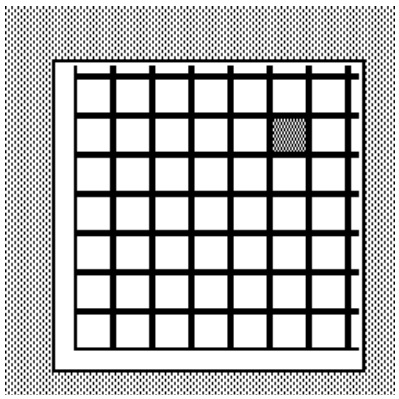

Figure 3-10. Example of SHADEGRIDBOX

3.7.3 OBTAINING GRID COORDINATES

You may obtain the grid coordinates closest to a specified X- or Y-coordinate using the functions **GRIDXCOORD** and **GRIDYCOORD**, which take the form:

Function:	GRIDXCOORD GRIDYCOORD
# Arguments:	2
Arguments:	1) XCOORD (YCOORD), an X- (Y-) coordinate 2) GRIDSPEC, a grid specification
Value:	The grid X-coordinate (Y-coordinate) closest to the specified point.

These functions return the x-coordinate (respectively, Y-coordinate) of the grid rectangle closest to the specified X- and Y-coordinates. Consider the following examples (where I've placed the cursor in the gridded region depicted in Figure 3.11):

```
<-(SETQ LMX (CURSORPOSITION NIL W1):XCOORD)
155

<-(GRIDXCOORD LMX gridspec)
6

<-(GRIDYCOORD LMX gridspec)
2
```

which by visual inspection, I can assure you is the case.

3.7.3.1 Getting a Grid Position

Rather than obtaining the X- and Y-coordinates individually, you may wish to obtain a grid position closest to a specified (X,Y) position. Let us define the function GRIDPOSITION:

Function:	GRIDPOSITION
# Arguments:	3
Arguments:	1) XorPOS, an X-coordinate or position

2) Y, a Y-coordinate
3) GRIDSPEC, a grid specification

Value: A position representing the grid coordinates closest to (X,Y).

We can define GRIDPOSITION as follows:

```
<-(DEFINEQ      (gridposition (xorpos y gridspec)
      (POINT
            (GRIDXCOORD
                  (COND
                        ((POSITIONP xorpos) xorpos:xcoord)
                        (T   xorpos))
                  gridspec)
             (GRIDYCOORD
                  (COND
                        ((POSITIONP xorpos) xorpos:ycoord)
                        (T   y))
                  gridspec))
   ))
(GRIDPOSITION)
```

Consider the following examples:

```
<-(GRIDPOSITION (CURSORPOSITION NIL W1) NIL gridspec)
(4 . 8)
```

This function is not included in the standard Interlisp sysout.

3.7.3.2 Getting the Grid Position of the Mouse

An alternative function, **GRIDPOSITION.OF.CURSOR**, allows you to obtain the grid position associated with the current cursor position. It takes the form:

Function: GRIDPOSITION.OF.CURSOR
Arguments: 1
Arguments: 1) GRIDSPEC, a grid specification
Value: A grid position closest to the current cursor position.

We can define GRIDPOSITION.OF.CURSOR as follows:

```
<-(DEFINEQ (gridposition.of.cursor (gridspec)
   (GRIDPOSITION (CURSORPOSITION NIL W1) gridspec)
   ))
```

Note that this function assumes the current display stream as the source for determining the coordinates of the cursor. I leave it as an exercise for the reader to redefine this function to accept a display stream.

Note that this function is not included in the standard Interlisp sysout.

3.7.4 OBTAINING SOURCE COORDINATES FROM GRID COORDINATES

You may obtain the (X,Y) coordinates of a grid position relative to the source display stream using the functions **LEFTOFGRIDCOORD** and **BOTTOMOFGRIDCOORD** which take the following form:

Function: LEFTOFGRIDCOORD

	BOTTOMOFGRIDCOORD
# Arguments:	2
Arguments:	1) GRIDX or GRIDY, an X- or Y-coordinate 2) GRIDSPEC, a grid specification
Value:	The X- or Y-coordinate of the grid rectangle at the specified grid coordinate.

LEFTOFGRIDCOORD returns the X-coordinate in source display stream coordinates of the left edge of a grid rectangle whose X-coordinate is GRIDX. BOTTOMOFGRIDCOORD performs analogously for the bottom edge of the grid rectangle for GRIDY. Consider the following examples:

```
<-(LEFTOFGRIDCOORD 4 gridspec)
100

<-(BOTTOMOFGRIDCOORD 7 gridspec)
175
```

3.7.5 GRID VARIABLES

The grid system uses several variables to validate the arguments specified in the functions. These are described in Table 3-10.

Table 3-10. Grid Variables

Variable	Usage
NORMALGRIDSQUARE	This contains the normal size for a grid square. Initially, its value is 16.
MINGRIDSQUARE	This variable specifies the minimum grid size. Initially, its value is 8.
MAXGRIDWIDTH	This is the maximum grid widh. Initially, its value is 199.
MAXGRIDHEIGHT	This is the maximum grid height. Initially, its value is 175.
GRIDTHICKNESS	This is the default width of the lines that separate the grid squares. Initially, its value is 2.

4. INPUT MANAGEMENT

Interlisp provides you with two types of input devices: a keyboard and a mouse. The keyboard has the standard complement of alphanumeric keys in QWERTY format as well as seven uninterpreted function keys. The ***mouse*** is a three button, hand-held device that controls the cursor on the display screen.

Interlisp-D separated the management of the display screen from the keyboard and the mouse. Because the keyboard and the mouse were managed entirely by Interlisp-D, many low level functions were directly accessible by the programmer.

4.1 MOUSE MANAGEMENT

`The mouse is used in two ways. First, it controls the position of the cursor on the screen. By moving the mouse, the cursor tracks its movement on the screen. Thus, you can use the mouse to point to various objects on the screen. Because the screen is addressed through pairs of X,Y coordinates, you can determine the exact location of the cursor at any time. The second use of the mouse is to select items from menus which are associated with one of the three mouse buttons. Menus allow you to parameterize the functionality of the system according to the current state of the computation.

4.1.1 USING THE MOUSE BUTTONS

The mouse buttons are commonly referred to as LEFT, MIDDLE, and RIGHT. These are the names that you would use when specifying arguments to various functions associated with the mouse and menus. Each mouse button may have different functions associated with it. Certain Interlisp subsystems support conventions concerning the use of the mouse buttons (for example, see the Window Manager).

When you create a subsystem under Interlisp, it is best to establish a convention for how the mouse buttons will be used within the subsystem. Interlisp has defined several conventions a priori such as the right mouse button always selects a menu of window level operations.

4.1.2 TESTING THE MOUSE STATE

You may test the current state of the mouse using **MOUSESTATE** which takes the form:

Function:	MOUSESTATE
# Arguments:	1
Arguments:	1) BUTTONFORM, an expression describing the states of the mouse to be tested
Value:	T, if the current mouse state corresponds to BUTTONFORM.

MOUSESTATE is a macro. The definition for MOUSESTATE is:

```
(PROGN
   '(GETMOUSESTATE)
   '(MOUSESTATE-EXPR (CAR ARGS) T))
```

where MOUSESTATE-EXPR is an internal function.

MOUSESTATE reads the current mouse state and compares it to the value of BUTTONFORM. It returns T if the current mouse state corresponds to the value of BUTTONFORM. BUTTONFORM takes one of the values described in Table 4-1.

Table 4-1. BUTTONFORM Values

Value	Meaning
LEFT, MIDDLE, RIGHT	Indicated that the specified key is being pressed.
UP	Indicated all keys are in the up state (e.g., unpressed).
ONLY <key>	Indicated only the specified key was pressed.
Boolean Expression	An expression using AND, NOT, and OR with the key designators.

Consider the following examples:

```
<-(MOUSESTATE LEFT)
T
```

while holding the left mouse button down.

```
<-(MOUSESTATE)
```

causes a Break Window to open with a message as depicted below.

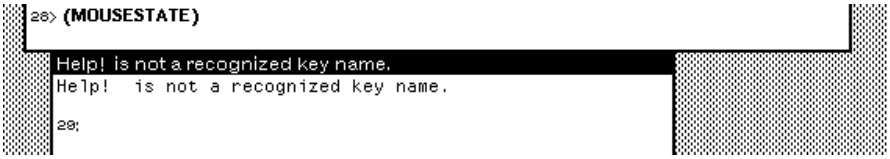

Figure 4-1. NIL supplied to MOUSESTATE

The macro **KEYSETSTATE** takes the same form as MOUSESTATE. However, it will also check the state of the five-finger keyset as well as the mouse buttons. The keys of the five-finger keyset are labeled LEFTKEY, LEFTMIDDLEKEY, MIDDLEKEY, RIGHTMIDDLEKEY, and RIGHTKEY.

4.1.3 TESTING THE LAST MOUSE STATE

You may test the last state of the mouse buttons using **LASTMOUSESTATE**:

Function:	LASTMOUSESTATE
# Arguments:	1
Arguments:	1) BUTTONFORM, an expression describing the states of the mouse to be tested
Value:	T, if the last mouse state corresponds to the value of BUTTONFORM.

LASTMOUSESTATE is a macro. Because it is a macro, its argument is not quoted. It is defined as:

```
<-(GETPROPLIST 'LASTMOUSESTATE)
(ARGNAMES    (BUTTONFORM)
    MACRO
        (ARGS (MOUSESTATE-EXPR (CAR ARGS) T))
)
```

LASTMOUSESTATE tests the value of the system variable LASTMOUSEBUTTONS rather than the current state of the mouse. BUTTONFORM has the same values as described in Section 4.1.2. LASTMOUSESTATE is usually used to determine which keys cause the button form used in MOUSESTATE to become true. Consider the following examples:

<-(LASTMOUSESTATE)

causes a Break Window to open with message as depicted in Figure 4-1 above.

<-(LASTMOUSESTATE LEFT)
NIL

The macro **LASTKEYSETSTATE** takes the same form as LASTMOUSESTATE. However, it will also check the state of the five-finger keyset as well as the state of the mouse buttons. MOUSESTATE-EXPR is an internal Interlisp function which reads the state of the mouse from the hardware registers.

4.1.4 WAITING UNTIL A MOUSE STATE BECOMES TRUE

You may insert a pause in your program which is controlled by the mouse keys using **UNTILMOUSESTATE**:

Function:	UNTILMOUSESTATE
# Arguments:	2
Arguments:	1) BUTTONFORM, an expression describing the states of the mouse to be tested 2) INTERVAL, the time to wait
Value:	T, if BUTTONFORM is satisfied before INTERVAL expires.

UNTILMOUSESTATE is a macro which waits until BUTTONFORM becomes true by continually testing the state of the mouse. BUTTONFORM takes the same values as described in Section 4.1.2. INTERVAL determines the duration of the pause. If NIL, Interlisp waits indefinitely. When INTERVAL expires and the BUTTONFORM has not been satisfied, UNTILMOUSESTATE returns NIL. INTERVAL is measured in milliseconds. Consider the following examples:

<-(UNTILMOUSESTATE (LEFT RIGHT))
T

when I press both the left and right mouse buttons. If no arguments are provided to UNTILMOUSESTATE, it opens a Break Window as depicted in Figure 4-1:

<-(UNTILMOUSESTATE)

4.3.4.1 Definition of UNTILMOUSESTATE

The definition of UNTILMOUSESTATE is:

```
<-(GETPROPLIST 'UNTILMOUSESTATE)
(ARGNAMES     (BUTTONFORM INTERVAL
(MACRO
    (ARGS
        (COND
          ((AND (CDR ARGS) (CADR ARGS) (NEQ (CADR ARGS) T))
          (* The time interval is specified and is not T or NIL; compile in time keeping loop)
           (LIST   'PROG
                   (LIST
                          (LIST    'TIMEOUT
                                   (LIST    'IPLUS
                                            '(CLOCK 0)
                                            (LIST 'OR (LIST 'NUMBERP (CADR ARGS)) 100)
                     ))
```

```
                    '(NOWTIME (CLOCK 0))
                )
                (QUOTE LP)
                (LIST 'COND
                      (LIST    (CONS (QUOTE MOUSESTATE) (LIST (CAR ARGS) T))
                               (QUOTE (RETURN T))
                ))
                (QUOTE
                    (COND
                          ((IGREATERP (CLOCK0 NOWTIME) TIMEOUT)
                           (RETURN NIL))
                    (T          \BACKGROUND)))
                 (QUOTE (GO LP))
          ))
          (T
                (LIST          'PROG NIL
                               (QUOTE LP)
                               (LIST    'COND
                                        (LIST (CONS 'MOUSESTATE (LIST (CAR ARGS) T))
                                         (QUOTE (RETURN T))
                ))
                 (QUOTE \BACKGROUND))
       (QUOTE (GO LP)))))
))
```

UNTILMOUSESTATE allows you to insert a pause that permits the user to think about the operation to be performed.

4.2 Low Level Access to the Mouse

A number of variables are maintained by Interlisp which reflect the state of the mouse (and its previous state) at any instant. A number of functions are provided for accessing the current position of the mouse and the state of its buttons.

4.2.1 Getting the Cursor's Position

The cursor tracks the movement of the mouse on the screen. In many cases, you will want to read the current position of the cursor on the screen. Two functions, **LASTMOUSEX** and **LASTMOUSEY**, allow you to read the X and Y coordinates, respectively, of the cursor with regard to the given display stream. They take the form:

Function:	LASTMOUSEX LASTMOUSEY
# Arguments:	1
Arguments:	1) DISPLAYSTREAM, the address of a display stream
Value:	The X (respectively Y) coordinate of the cursor.

LASTMOUSEX and LASTMOUSEY return the X and Y coordinates of the cursor in the coordinates of the DISPLAYSTREAM. Consider the following example:

```
<-(LASTMOUSEX)
298

<-(LASTMOUSEY)
-38
```

At this time, the current display stream was the Interlisp Executive window and the cursor was residing below the window about two-thirds across its width. Thus, -38 is the Y coordinate of the mouse relative to the most recent display stream.

Negative coordinates returned from LASTMOUSEX and LASTMOUSEY indicate that the mouse lies outside the display stream. In this case, because DISPLAYSTREAM was NIL, the display stream used internally defaulted to the Interlisp Executive Window.

It is recommended that you use these functions to locate the cursor with respect to the display stream coordinates and pass the resulting values to a function.

4.2.2 DECODING THE MOUSE BUTTONS

You may read the names of the buttons which are currently pressed using the function **DECODEBUTTONS**:

Function:	DECODEBUTTONS
# Arguments:	1
Arguments:	1) BUTTONSTATE, an integer describing the buttons to be tested
Value:	A list of the buttons which are currently pressed.

DECODEBUTTONS returns a list of the buttons which are currently pressed that corresponds to the description given in BUTTONSTATE. It can return the names of the mouse buttons or the five-finger keyset buttons.

4.2.3 GETTING THE MOUSE STATE

You can obtain the current mouse state using the function **GETMOUSETSTATE**:

Function:	GETMOUSESTATE
# Arguments:	0
Arguments:	N/A
Value:	NIL

GETMOUSESTATE sets the values of the mouse variables which are described in the following section. Consider the example:

```
<-(GETMOUSESTATE)
NIL

<-LASTMOUSEX
262

<-LASTMOUSEY
464

<-LASTMOUSEBUTTONS
0

<-LASTKEYBOARD
128
```

4.2.3.1 Redefining Keys for Mouse Buttons

There are instances where you may want to redefine keyboard keys as mouse buttons (such as when a mouse button breaks). You may advise GETMOUSESTATE to simulate a mouse button as follows:

```
(ADVISE     'GETMOUSESTATE
            'AFTER
            ()
            '(AND       (KEYDOWNP 'AGAIN)
                        (SETQ LASTMOUSEBUTTONS  (LOGOR LASTMOUSEBUTTONS 2))
            ))
```

which makes the AGAIN key act like the right mouse button. This piece of code was devised by Mayank Prakash at MCC.

4.2.4 CONFIRMING AN OPERATION WITH THE MOUSE

You may confirm selections or operations with the mouse (i.e., by pressing the left mouse button) using the function **MOUSECONFIRM**:

Function:	MOUSECONFIRM
# Arguments:	4
Arguments:	1) PROMPTSTRING, a string to be displayed requesting confirmation
	2) HELPSTRING, a string displayed if help is requested
	3) WINDOW, a window handle
	4) DON'TCLEARWINDOWFLG, a flag
Value:	T

MOUSECONFIRM displays the prompt message, PROMPTSTRING, in the specified window. Typically, this will be the prompt window, but you may specify any other window that you wish. If WINDOW is NIL, it defaults to the Interlisp Executive Window. Consider the following example:

```
<-(MOUSECONFIRM "Delete File:" NIL PROMPTWINDOW)
T
```

When I execute this expression, MOUSECONFIRM clears the specified window - in this case, the prompt window - and displays PROMPTSTRING. It then changes the cursor to resemble the MOUSECONFIRMCURSOR. The shape of the mouse confirm cursor indicates that you are to press the left mouse button in order to confirm the operation specified by the prompt string. Pressing the left mouse button causes the value T to be returned as the value of MOUSECONFIRM. Any other mouse button causes NIL to be returned. Figure 4.2 depicts the shape of the MOUSECONFIRMCURSOR.

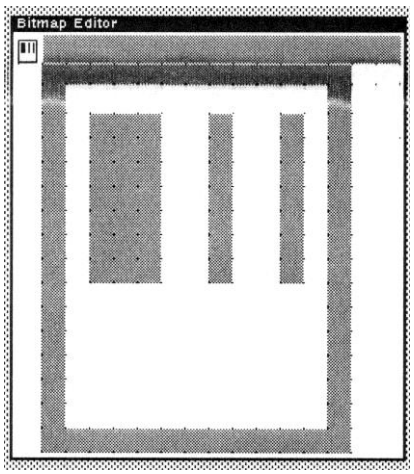

Figure 4-2. Example of the MOUSECONFIRMCURSOR

Note that if WINDOW is not open, but is a valid window handle, the window will be opened and the prompt string displayed in the window.

If DON'TCLEARWINDOWFLG is T, the window will not be cleared before the prompt string is displayed.

4.2.5 MOUSE SYSTEM VARIABLES

Interlisp maintains several variables which record the state of the mouse as presented in Table 4-2.

Table 4-2. Mouse System Variables

Variable	Usage
LASTMOUSEX	The X position of the cursor in absolute screen coordinates.
LASTMOUSEY	The Y position of the cursor in absolute screen coordinates.
LASTMOUSEBUTTONS	An 8-bit integer whose bits correspond to the mouse buttons that are currently pressed. The buttons are described as follows: Value Meaning 4Q left mouse button 2Q right mouse button 1Q middle mouse button
LASTKEYBOARD	An 8-bit integer recording the state of certain keys on the keyboard, as presented in Table 4-5.
LASTMOUSETIME	The time, in milliseconds, since the mouse state was last read. This variable is a 16-bit integer which rolls over every 65+ seconds.

And, for the keyset buttons.

Table 4-3. Values of LASTMOUSEBUTTONS

Value	Usage
200Q	left keyset button

100Q	leftmiddle keyset button
40Q	middle keyset button
20Q	rightmiddle keyset button
10Q	right keyset button

Table 4-4. Values of LASTKEYBOARD

Value	Meaning
200Q	lock key
100Q	left shift
40Q	crtl
10Q	right shift
4Q	blankBottom
1Q	blankTop

4.3 CURSOR MANAGEMENT

The cursor is a symbol which is displayed on the screen which addresses specific bits of the display screen. The cursor is moved by moving the mouse. You may edit the cursor bit map to design your own cursors. Many applications will design new cursors for different subsystems or functions provided by the application in order to provide you with meaningful information on the state of the program. As you enter a new subsystem, the shape of the cursor changes.

4.3.1 REPRESENTING THE CURSOR

A cursor is represented by a record which has the definition:

```
((CURSORBITMAP .       CURSORHOTSPOT)
 (CURSORHOTSPOT <- (create POSITION)
 (ACCESSFNS
       ((CURSORHOTSPOTX
              (fetch (POSITION XCOORD) of  (fetch (CURSOR CURSORHOTSPOT) of DATUM))
              (replace (POSITION XCOORD) of (fetch (CURSOR CURSORHOTSPOT) of DATUM))
                     with NEWVALUE))
       ((CURSORHOTSPOTY
              (fetch (POSITION YCOORD) of (fetch (CURSOR CURSORHOTSPOT) of DATUM))
              (replace (POSITION YCOORD) of (fetch (CURSOR CURSORHOTSPOT) of DATUM))
                     with NEWVALUE))))
 (TYPE?
    (AND
       (type? BITMAP  (fetch (CURSOR CURSORBITMAP) of (LISTP DATUM)))
       (type? POSITION (fetch (CURSOR CURSORHOTSPOT) of DATUM))))
 (SYSTEM))
```

An example of a cursor record (with sample values) is:

```
CURSORHOTSPOTX     0
CURSORHOTSPOTY     15
CURSORHOTSPOT      (0 . 15)
CURSORBITMAP       {BITMAP}#74,124644
```

The cursor bitmap is displayed on the screen by bit-blitting the cursor bit map to the display screen bit map. The standard Interlisp cursor bit map has the following appearance:

<-(EDITBM (CURSORBITMAP))
{BITMAP}#70,167770

The area of the cursor is defined by two variables: CURSORWIDTH and CURSORHEIGHT.

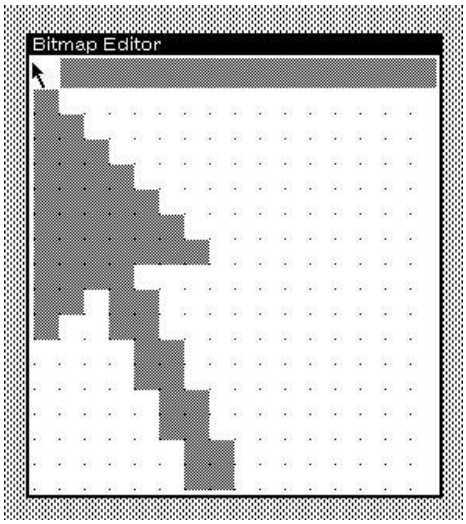

Figure 4-3. Standard Cursor Bitmap

The cursor has a location called the ***hot spot***. This hot spot is a point position within the cursor bit map whose coordinates are returned when the cursor is position is queried.

4.3.1.1 Accessing the Current Cursor's Bitmap

The current cursor is represented by a bitmap which may be obtained by executing the function **CURSORBITMAP**, which takes the form:

Function:	CURSORBITMAP
# Arguments:	0
Arguments:	N/A
Value:	A bitmap handle.

Consider the following example:

<-(CURSORBITMAP)
{BITMAP}#70,167770

which, in this case, represents the bitmap of the standard cursor.

This function is particularly useful for checking modes if you switch cursors to indicate different modes of interaction to the user. By checking the cursor bitmap, you can determine which mode you are in.

4.3.1.2 Getting the Cursor's Hotspot Coordinates

Because you can change the coordinates of the cursor's hotspot, you also need to be able to access those coordinates to determine if the cursor is actually pointing at something. You can use the function **CURSORHOTSPOT**:

Function:	CURSORHOTSPOT
# Arguments:	1
Arguments:	1) POSITION, the coordinates of a new hotspot
Value:	A position representing the cursor's hotspot.

This function retrieves the hotspot coordinates of the current system cursor. Consider the following examples:

```
<-(CURSORHOTSPOT)
(0 . 15)

<-(CURSORHOTSPOT (POINT 5 5))
(0 . 15)

<-(CURORHOTSPOT)
(5 . 5)
```

In this example, I also changed the coordinates of the hotspot of the current system cursor (the arrow) to midway down the length of the arrow.

In general, you must be careful about changing the hotspot of a cursor to some obvious feature. Otherwise, as the user tries to position the cursor near an object, they may make the wrong selection. It is personally frustrating to see the cursor positioned on an object but not know where its hot spot is.

4.3.1.3 Interlisp Cursors

Interlisp defines several cursors that indicate specific conditions to be observed or subsystems in operation as described in the following table.

Table 4-5. Interlisp Cursors

Cursor	Usage
Waiting Cursor	The ***waiting cursor*** takes the form of an hour glass. It indicates that a long computation is in progress by the system.
Garbage Collection Cursor	The ***Garbage Collection Cursor*** indicates that Interlisp is currently sweeping your virtual memory to gather garbage and place it on the free space lists.
Saving Cursor	The ***Saving Cursor*** indicates that Interlisp is saving a copy of your virtual memory to the external disk
SYSOUT Cursor	The ***sysout cursor*** indicates that Interlisp is creating a sysout for you.

The Waiting Cursor

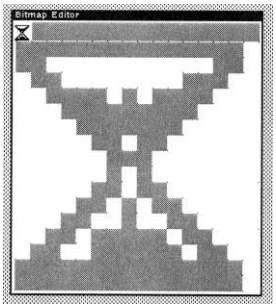

Figure 4-4. The Waiting Cursor

The Garbage Collection Cursor

The ***Garbage Collection Cursor*** indicates that Interlisp is currently sweeping your virtual memory to gather garbage and place it on the free space lists.

The Saving Cursor

The ***Saving Cursor*** indicates that Interlisp is saving a copy of your virtual memory to the external disk. It is depicted in Figure 4-5.

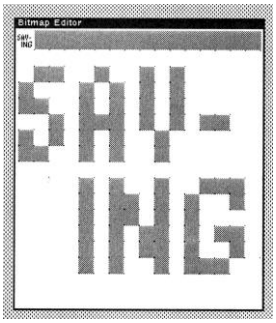

Figure 4-5. The Saving Cursor

The SYSOUT Cursor

The ***sysout cursor*** indicates that Interlisp is creating a sysout for you.

4.3.2 CREATING A CURSOR

You may create a new cursor object using the function **CURSORCREATE**:

Function:	CURSORCREATE
# Arguments:	3
Arguments:	1) BITMAP, a bitmap of the new cursor

	2) X, the X coordinate of the hot spot of the cursor
	3) Y, the Y coordinate of the hot spot of the cursor
Value:	A cursor handle.

CURSORCREATE creates a new cursor object. BITMAP is the representation of the cursor which will be displayed on the display screen. The position (X . Y) is the ***hot spot*** coordinates of the cursor within the bit map. X and Y are subject to the following conditions:

1. 0 < X < CURSORWIDTH
2. 0 < Y < CURSORHEIGHT

If X is NIL, it defaults to 0. If Y is NIL, it defaults to CURSORHEIGHT-1. Consider the following example:

```
<-(SETQ SHKCURSOR (CURSORCREATE (BITMAPCREATE 16 16 1) 0 15))
({BITMAP}#57,171504 0 . 15)
```

This cursor contains my initials. The cursor was created by editing the bitmap.

4.3.3 OBTAINING THE CURSOR POSITION

The cursor position may be obtained using the function **CURSORPOSITION**:

Function:	CURSORPOSITION
# Arguments:	3
Arguments:	1) NEWPOSITION, a new position for the cursor
	2) DISPLAYSTREAM, a display stream handle
	3) OLDPOSITION, a position variable
Value:	The value of the old position of the cursor.

CURSORPOSITION retrieves the old position of the cursor relative to the coordinate system of the specified display stream. If DISPLAYSTREAM is NIL, it is determined relative to the current display stream (usually, the window most recently written to). If OLDPOSITION is a position, the cursor position will be assigned to it. Consider the following example:

```
<-(CURSORPOSITION)
(489 . 110)
```

which is relative to the Interlisp Executive Window because it is the last window activated.

```
<-(CURSORPOSITION NIL (PROMPTWINDOW))
(262 . -257)
```

where the cursor is positioned away from the prompt window.

Consider the region AREGION = (200 200 600 600). When we attempt to test against a region:

```
<-(CURSORPOSITION NIL AREGION)
FILE NOT OPEN
(200 200 600 600)
```

but, testing against the window AWINDOW = (CREATEW AREGION):

```
<-(CURSORPOSITION NIL AWINDOW)
(297 . 273)
```

If NEWPOSITION is non-NIL and is a position, the cursor will be placed at the position indicated by NEWPOSITION relative to the specified display stream or the current display stream. Consider the following examples:

```
<-(CURSORPOSITION)
(490 . 109)

<-(CURSORPOSITION (POINT 300 300) AWINDOW)
(297 . 273)

<-(CURSORPOSITION)
(493 . 136)

<-(CURSORPOSITION (POINT 400 400) AWINDOW)
(593 . 236)
```

4.3.4 ADJUSTING THE CURSOR POSITION

You may adjust the current position of the cursor by specifying the number of pixels to move in either the X or Y directions or both. The adjustments may be positiove or negative. Typically, they are integer values. You use the function **ADJUSTCURSORPOSITION**:

Function:	ADJUSTCURSORPOSITION
# Arguments:	2
	Arguments: 1) DELTAX, adjustment for the X axis
	2) DELTAY, adjustment for the Y axis
Value:	NIL.

ADJUSTCURSORPOSITION moves the cursor DELTAX pixels in the X direction and DELTAY pixels in the Y direction. If either DELTAX or DELTAY is NIL, it defaults to 0. Consider the following example:

```
<-(CURSORPOSITION)
(1046 . -65)

<-(ADJUSTCURSORPOSITION -100 50)
NIL

<-(CURSORPOSITION)
(946 . -15)
```

Note that the cursor position is given in absolute screen coordinates relative to the window which currently has control of the mouse. In these examples, this is the Interlisp Executive Window. If either DELTAX or DELTAY or both is large enough to move the cursor beyond the boundaries of the current display stream or window. Consider the following example:

```
<-(CURSORPOSITION)
(335 . 69)

<-(ADJUSTCURSORPOSITION 2000 1000)
NIL
```

moves the cursor to the upper right hand corner of the display screen. The presumed cursor position obtained by adding 2000 to 335 and 1000 to 69 would place the cursor outside the physical boundaries of the screen. So, Interlisp moves the cursor to the farthest physical boundary of the screen.

4.3.5 COPYING A CURSOR RECORD

A copy of the record describing the current cursor may be obtained using the function **CURSOR**:

Function:	CURSOR
# Arguments:	1
	Arguments: 1) NEWCURSOR, a cursor handle
Value:	The handle of the old cursor.

CURSOR, without any arguments, returns a copy of the record describing the current cursor. Consider the following example:

```
<-(CURSOR)
({BITMAP}#65,123644 0 . 15)
```

If NEWCURSOR is a cursor record instance, the cursor will be set to the values specified by NEWCURSOR. Consider the following example:

```
<-(CURSOR shkcursor)
({BITMAP}#57,171746 0 . 15)
```

which is the record associated with the old cursor.

If NEWCURSOR is T, the cursor is set to the Interlisp default cursor whose handle is the value of DEFAULTCURSOR. Consider the following example:

```
<-(CURSOR T)
({BITMAP}#57,171272 0 . 15)
```

which causes the cursor to again become the arrow pointing to the upper left corner.

4.3.6 SETTING THE CURSOR

You may set the cursor by using the function **SETCURSOR**:

Function:	SETCURSOR
# Arguments:	1
` Arguments:	1) NEWCURSOR, a cursor handle
Value:	The X coordinate of the hot spot.

SETCURSOR acts just like CURSOR in establishing the features of the cursor except that it does not return the handle of the old cursor. Thus, it does not use any storage. Consider the following examples:

```
<-(SETCURSOR)
ILLEGAL ARG
NIL

<-(SETCURSOR DEFAULTCURSOR)
0

<-(SETCURSOR shkcursor)
0
```

4.3.7 INVERTING THE CURSOR

The cursor is normally displayed as a black-on-white bitmap. You may invert the cursor bit map using the function **FLIPCURSOR**:

Function:	FLIPCURSOR
# Arguments:	0
Arguments:	N/A
Value:	NIL

FLIPCURSOR inverts the cursor as it is displayed on the screen. It does not invert the bitmap used to create the cursor.

Inverting the cursor is a useful way to indicate to the user that he or she should wait for an operation to finish or that some background operation is being performed by the system. For example, when the standard cursor inverts, the systems has invoked the garbage collection subsystem. Figure 4-6 displays an image of the standard cursor when it is inverted.

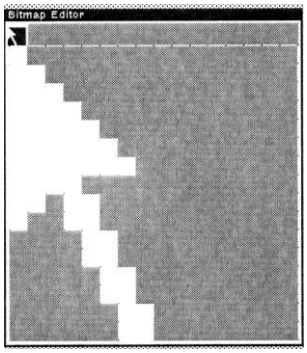

Figure 4-6. An Inverted Standard Cursor

4.3.8 ALERTING THE USER

In many applications, you need to catch the user's attention when a message is displayed to which he or she should respond. One way to do this is to flip the cursor rapidly as a means of signaling that he or she should look for a message. Let us define a simple function **ALERT.USER**, which prints a message in the prompt window and flashes the cursor at the user. ALERT.USER may be defined as:

```
<-(DEFINEQ   (alert.user (msg #times)
      (AND  msg (PROMPTPRINT msg))
      (FOR I FROM 1 TO #times
            DO (FLIPCURSOR))
    ))
(ALERT.USER)
```

4.3.9 SAVING CURSORS IN A FILE

Once you have created a cursor, you may save it for later use on a file using the CURSORS file package command. Consider the following example:

4.4 PROMPTFORWORD

PROMPTFORWORD is a function that reads an arbitrary sequence of characters from the keyboard without involving the READ function syntax. It provides many optional features through a large number of arguments. However, the default value for most of these options is NIL so that simple calls can easily be made. When using PROMPTFORWORD, you may supply a prompt string to initiate the data entry. You may also supply a candidate string which is printed and which becomes the default response if a word terminator is typed or a timeout occurs.

4.4.1 USING PROMPTFORWORD

A call to PROMPTFORWORD takes the following form:

Function:	PROMPTFORWORD
# Arguments:	8
Arguments:	1) PROMPT.STR, the prompt string
	2) CANDIDATE.STR, the candidate default answer
	3) GENERATE?LIST.FN, a help function or string
	4) ECHO.CHANNEL, A display stream to echo input
	5) DONTECHOTYPEIN.FLG, inhibits echoing of input
	6) URGENCY.OPTION, the time dureation to wait for input
	7) TERMINCHARS.LST, a list of word terminator character codes
	8) KEYBD.CHANNEL, a display stream for receiving input
Value:	The input string, if any.

As you can see, PROMPTFORWORD provides you with considerable flexibility in receiving input from the user. The individual arguments will be discussed in the sections below.

The default input and echo streams are both from the keyboard. The terminal table in effect during type-in allows most control characters to be INDICATE'd.

PROMPTFORWORD returns NIL if you type a null string. Consider the following example:

```
<-(PROMPTFORWORD)<CR>
NIL
```

NIL is returned when no candidate string is given and you type a word terminator such as a carriage return. Normally, PROMPTFORWORD returns a string. Consider the following example:

```
<-(PROMPTFORWORD "Enter name:")
Enter Name: Steve
"Steve"
```

4.4.1.1 The Prompt String

Usually, you will need to alert the user that input is expected. The argument PROMPT.STR is a string which is printed when PROMPTFORWORD is executed. If PROMPT.STR is not a string, PROMPTFORWORD coerces it to a string via MKSTRING. When the string is printed, an additional space is printed to separate the prompt string from the answer. Consider the following example:

```
<-(PROMPTFORWORD "Enter file name:")
Enter file name: X.TXT<CR>
"X.TXT"
```

It is suggested that you terminate the prompt string with a punctuation mark that is usually associated with queries, such as ? or :.

4.4.1.2 The Candidate String

You may specify a default answer as the value of the argument CANDIDATE.STR. After the prompt string, if any, is printed, the candidate string will be printed. This string is offered as the initial contents of the input buffer. If you type any characters other than word terminator characters, the candidate string will be erased and the new input becomes the contents of the input buffer. Consider the following example:

```
<-(PROMPTFORWORD "Enter file name:" "SHK.LISP")
Enter file name: SHK.LISP<CR>
"SHK.LISP"
```

which shows the default behavior. Now, consider the example:

```
<-(PROMPTFORWORD "Enter file name:" "SHK.LISP")
Enter file name: TOM.LISP<CR>
"TOM.LISP"
```

The cursor is placed at the end of the default string. You may move it via the mouse or by pressing the backspace key. Then, you may retype any characters that you wish to change.

4.4.1.3 Providing Help

Additional help may be invoked during type-in by entering the character ?. You may provide additional help to the user in one of two ways:

1. An additional help string
2. A function which is applied to PROMPT.STR and CANDIDATE.STR to generate a list of potential candidates

Either of these values is assigned to the argument GENERATE?LIST.FN. If GENERATE?LIST.FN is a function (typically, a LAMBDA expression), it is given the strings PROMPT.STR and CANDIDATE.STR as its arguments. It should return a list of potential candidates which will be printed on a separate line. More likely, this function will return NIL, but will independently print the list of potential candidates in some useful format. After the candidates are printed, the prompt is restarted and any type-in is re-echoed. Consider the following examples:

```
<-(PROMPTFORWORD        "Select a color"
                        "Red"
                        (FUNCTION
                              (LAMBDA (PROMPT CANDIDATE)
                                        (LIST "Yellow" "Orange" "Blue"))
                        ))
Select a color: Red?<CR>
{Yellow Orange Blue}
Select a color: Red<CR>
Red
```

If GENERATE?LIST.FN is a function, its value is cached so that it will be executed at most once per call to PROMPTFORWORD.

4.4.1.4 The Echo Channel

Usually, input is echoed to the terminal. Thus, the value NIL for this argument defaults to T, the terminal display stream. normally, this output stream will be the value returned by TTYDISPLAYSTREAM.

However, you may direct that the characters typed-in be echoed to another display stream by assigning a display stream handle as the value of ECHO.CHANNEL. If you want to echo the input to the current output stream (whatever it is!), you should use the expression (GETSTREAM NIL 'OUTPUT) as the value of the argument ECHO.CHANNEL. The display stream where the next input will be echoed has a flashing caret to indicate where the input will be displayed.

4.4.1.5 Inhibiting Echoing of Input

The echoing of characters typed-in by the user may be inhibited by assigning T as the value of the argument DONTECHOTYPEIN.FLG. Consider the following example:

```
<-(PROMPTFORWORD "Enter name:" "Steve" NIL NIL T)
Enter name: "Steve"
```

When the characters "Steve" were typed, they were not echoed on the input screen. Once a <CR> was typed, PROMPTFORWORD displayed "Steve" as the value it returns.

If the value of DONTECHOTYPEIN.FLG is a single character atom or string, its value will be echoed instead of the actual input. The IRM notes that LOGIN will prompt you for a password with the value of DONTECHOTYPEIN.FLG set to "*".

4.4.1.6 Input Wait Time

You may specify how long PROMPTFORWORD will wait for the user to type in a response to the query by assigning a value to the argument URGENCY.OPTION.

If URGENCY.OPTION is NIL, PROMPTFORWORD will essentially wait "forever" for the user to enter data, just as READ does. Alternatively, if it is T, PROMPTFORWORD waits "forever", but periodically will flash the window associated with the display stream to alert the user that input is expected.

If URGENCY.OPTION is a number, it specifies the number of seconds to wait for the user to type in a response. If time expires before the user types input, then CANDIDATE.STR is returned.

If URGENCY.OPTION is the atom TTY, then PROMPTFORWORD seizes the keyboard immediately. The cursor will be changed (temporarily) to a different shape (see below) to indicate that input is urgently requested.

4.4.1.7 The Word Terminator List

When you type input in response to a query from PROMPTFORWORD, the input must be terminated by a valid *word terminator*. The argument TERMINCHARS.LST is a list of character codes specifying those characters which are word terminators. The default value is (CHARCODE (EOL ESCAPE LF SPACE TAB)). It may also be a single character code.

Usually, you will use the default word terminator set for most input. However, for certain applications it may be both feasible and desirable to specify certain special characters to terminate segments of the input.

4.4.1.8 The Input Stream

Normally, input for PROMPTFORWORD will be taken from the keyboard input stream. However, you may specify an alternative input stream by assigning a non-NIL value to the argument KEYBD.CHANNEL.

Note that the terminal input stream T is a buffered keyboard input stream (see Section 14.1,I). It is not suitable for PROMPTFORWORD according to the IRM.

4.4.2 RESPONSE TO CONTROL CHARACTERS

PROMPTFORWORD will recognize several special characters as described in the following table:

Table 4-6: Special Characters

Character	Response
CTRL-A	Deletes the last character typed and erases it from the echo stream
BACKSPACE	Deletes the last character typed and erases it from the echo stream
DELETE	Deletes the last character typed and erases it from the echo stream
CTRL-Q	Erases all type-in so far
CTRL-R	Reprints the accumulated input string
CTRL-V	"Quotes" the next character so that it is added to the accumulated input string regardless of any special meaning it may have
CTRL-W	Erases the last word
?	Invokes the "help" facility whose response is determined by the value of the argument GENERATE?LIST.FN

5. WINDOW MANAGEMENT

A ***window*** is a rectangular area mapped onto the display screen through which you interact with a program. Typically, we speak of a program being "embedded" within the window.

Windows form the kernel of a multiprogramming paradigm. One window corresponds to one program. You can interact with or operate as many programs as you have windows present on the screen. Each program may be different: viewing a stack frame, executing a function, inspecting a data structure, etc.

Windows may overlap each other on the screen without the loss of information. Windows are ordered in depth from the user to the background. The currently active window will be highlighted by having its brought to the top of the occlusion stack. To activate another window, you merely place the cursor in any portion of the window and click the leftmost button. The window is brought to the top of the stack of windows.

5.1 WINDOW CHARACTERISTICS

Each window has a number of characteristics that define its identity and how it may be used.

5.1.1 WINDOW REPRESENTATION

A window is represented on the display screen as a rectangular area. Each window has a border of variable width. Interior to the border is the ***pane*** of the window where data may be displayed. When the cursor enters the pane, the mouse buttons may become active. The operations associated with the mouse buttons depend on the underlying functions attached to the window properties. At the top of the window is a ***title pane*** where the name of the window is displayed.

5.1.2 WINDOW STATES

Windows may exist in one of two states: open or closed. An open window is placed on Interlisp's occlusion stack. It will be visible on the display screen unless it is obscured by other windows. When a window is in the open state, it is susceptible to mouse operations. When a window is in a closed state, it is not visible nor represented on the screen. A window in the closed state cannot be operated upon by the mouse until it is opened. However, under program control, a window may be forced open by printing to it.

Even when a window is closed, its window handle is still defined in the system. You may reopen a closed window by referencing its window handle.

5.1.3 ICONS

An ***icon*** is a small rectangle containing text or a bitmap which identifies a shrunken window. The icon may be moved about the screen by placing the cursor in the icon and pressing the left mouse button. You may then drag the icon about the screen to the position that you want. Releasing the left mouse button deposits the icon at that screen position. The icon may be expanded to the window it represents by placing the cursor in the icon and clicking the middle mouse button. When an icon is created, it will be cached under the window property ICONWINDOW on the window with which it is associated. This permits repeated calls to SHRINKW and EXPANDW using the same icon.

5.2 WINDOW TYPES

When Interlisp is initialized from the baseline sysout, three windows will appear on the screen as depicted in Figure 5-1.

Table 5-1. Initial Interlisp Windows

Window	Description
Prompt Window	Displays help messages or information requests emitted by Interlisp, but may also be used by applications programs.
Interlisp Executive Window	Corresponds to the file T in the EXEC process where the top level read-eval-print loop operates.
Logo Window	Displays the Interlisp logo.

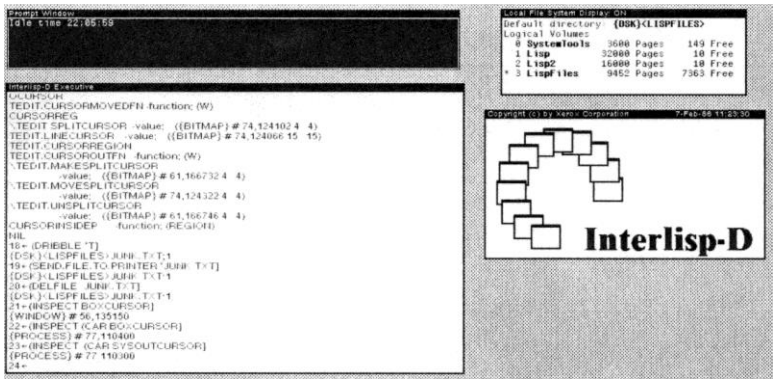

Figure 5-1. Initial Interlisp-D Windows

5.2.1 MANIPULATING THE PROMPT WINDOW

The *Prompt Window* is used by many of the Interlisp subsystems to display help messages or to request additional information through prompt messages. The identifier of the prompt window is bound to the global variable PROMPTWINDOW:

```
<-PROMPTWINDOW
{WINDOW}#74,25640
```

Of course, the address of the PROMPTWINDOW handle varies with the amount of memory you have in your system.

The prompt window is intentionally sized to be rather small. Its background shade is BLACKSHADE so that any messages that are printed to it appear in a white-on-black format. This is intended to catch you attention. You may resize the prompt window by selecting the **Shape** operation from the window operations menu (selected via the RIGHT mouse button).

5.2.1.1 Printing to the Prompt Window

An expression may be printed in the prompt window using the function **PROMPTPRINT**, which takes the following form:

Function:	PROMPTPRINT
# Arguments:	1
Arguments:	1) EXPRESSION, an expression
Value:	NIL.

EXPRESSION is evaluated and its value is displayed in the prompt window as depicted in Figure 5-2:

```
<-(PROMPTPRINT "Use the right button to select the window menu")
NIL
```

Note that the prompt window is cleared at each call to PROMPTPRINT. If you do not want to clear the prompt window when writing to it, then you should use the system variable PROMPTWINDOW as the second argument to one of the PRINx functions (see Section 15.1,I).

Figure 5-2. Printing to the Prompt Window

5.2.1.2 Clearing the Prompt Window

When a message is displayed in the prompt window, it remains until it is overwritten or the prompt window is cleared. You may clear the prompt window using the function **CLRPROMPT**:

Function:	CLRPROMPT
# Arguments:	0
Arguments:	none
Value:	NIL

CLRPROMPT erases the prompt window pane. You should clear the prompt window before each new prompt messgae is written to the window in order not to confuse the user.

5.2.2 MANIPULATING THE LOGO WINDOW

The Logo Window has the appearance depicted in Figure 5.-3. The identifier of the logo window is bound to the global variable LOGOW:

```
<-LOGOW
{WINDOW}#65,125234
```

Many application programs replace the Interlisp logo with a logo of their own design when they are loaded into memory.

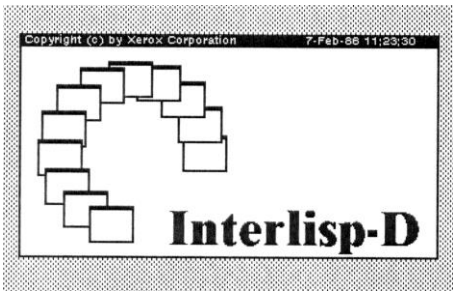

Figure 5-3. Interlisp Logo Window

Of course, the Logo window handle address varies with the amount of memory in your system.

5.2.2.1 Creating a Logo Window

A user may create his or her own logo window using the function **LOGOW**:

Function:	LOGOW

# Arguments:	4
Arguments:	1) STRING, the string to be printed
	2) WHERE, the position of the lower left corner of the window
	3) TITLE, the title of the window
	4) ANGLEDELTA, the angle of the boxes in the picture.
Value:	A window handle.

LOGOW creates a duplicate of the standard Interlisp logo window. However, it substitutes the value of STRING for the phrase "Interlisp". If STRING has the value NIL, the string displayed is "Interlisp". Figure 5-4 depicts a logo window created using my initials:

```
<-(LOGOW "SHK")
{WINDOW}#65,125150
```

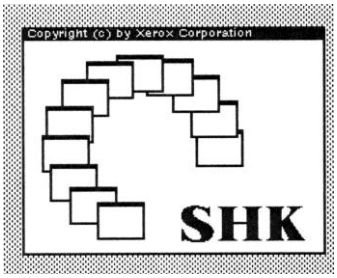

Figure 5-4. Example of a Custom Logo Window

WHERE specifies the coordinates of the lower left corner of the window in which the logo will be displayed. If WHERE has the value NIL, you will be prompted to specify a position via the mouse. Interlisp sizes the window to accommodate the boxes and the logo string.

TITLE is the title of the window. If NIL, it defaults to the Xerox copyright notice and date.

ANGLEDELTA is the angle (in degrees) or pitch between the boxes in the picture. If NIL, it defaults to 23 degrees. Examples of the boxes pitched at 10 and 40 degrees are shown in Figures 5-5 and 5-6.

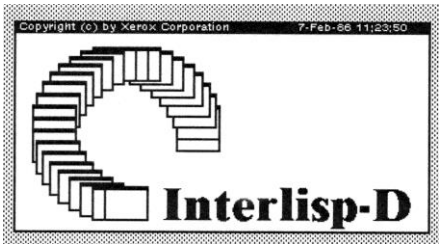

Figure 5-5. Logos with 10 Degree Pitch Angle

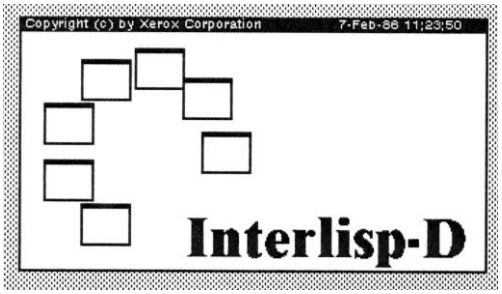

Figure 5-6. Logos with 40 Degree Pitch Angle

5.2.3 INTERLISP EXECUTIVE WINDOW

The major window through which you interact with Interlisp is called the Interlisp Executive Window (formerly the Top-Level Typescript Window). When you log onto Interlisp, the window is blank except for a number and left arrow which indicates the history event number. A user may type any Interlisp expression after the left arrow. It will be accepted by LISPXREAD as part of the "read-eval-print" loop. Figure 5-7 depicts the Interlisp Executive Window with some commands already executed in it.

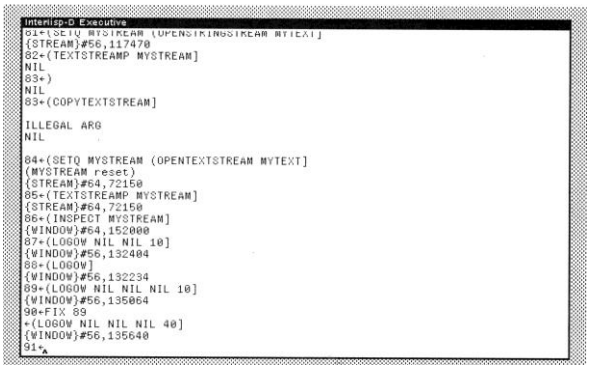

Figure 5-7. The Interlisp Executive Window

5.3 INTERACTIVE WINDOW OPERATIONS

A user may interactively manipulate windows on the display screen using the mouse to point to the window. A number of standard functions are defined for a window when it is created. These functions are accessed by placing the cursor in the window pane and pressing the rightmost mouse button. The standard *Window Menu* of window operations will appear at the location designated by the cursor. You may select one of the operations in the menu by moving the cursor until the operation is highlighted and, then, releasing the rightmost mouse button.

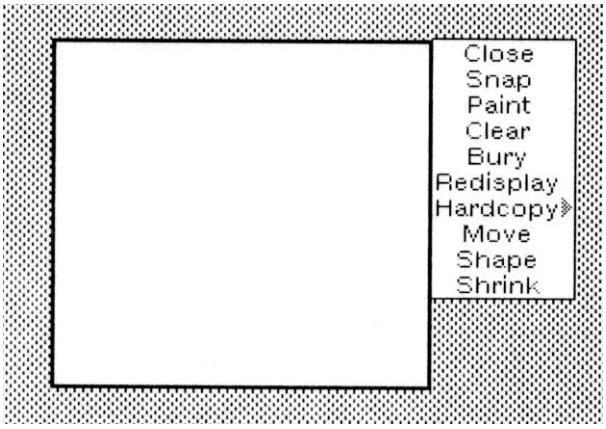

Figure 5-8. The Standard Window Menu

5.3.1 CLEARING A WINDOW

To clear a window, select the CLEAR operation from the window operation menu. When the mouse button is released, the contents of the window pane will be erased and the cursor will be repositioned to the upper lefthand corner. Clearing a window fills the window with its background shade. Usually, this will be WHITESHADE, but the background shade may be switched to some other texture.

5.3.2 CLOSING A WINDOW

A window may be closed by selecting the CLOSE operation from the window operation menu. When the mouse button is released, the window will be removed from the screen. The window is removed from the occlusion stack. Note that closing a window does not release its window handle. A closed window may be re-opened using the function OPENW.

5.3.3 BURYING A WINDOW

A user may bury a window by selecting the BURY operation from the window operation menu. When the mouse button is released, the window is placed at the bottom of the occlusion stack. Any windows that were obscured by the window that is buried will now be visible. Figure 5-9 depicts this operation where I selected the Bury operation for the Logo Window. It is now obscured behind an Inspector window.

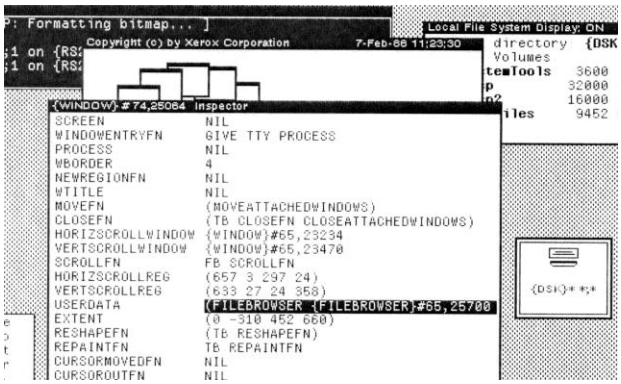

Figure 5-9. Burying a Window

5.3.4 MOVING A WINDOW

A window may be moved to another location on the screen by selecting the MOVE operation from the window operation menu. When the right mouse button is released, you should depress the left mouse button. At this time, a "ghost frame" of the same dimensions will appear on the screen. You can move the ghost frame by moving the mouse. When you have selected the location where the window is to be positioned anew (indicated by the location of the ghost frame), release the left mouse button. The window will be erased from its current location and appear at the new location.

5.3.5 SHAPING A WINDOW

A window may be reshaped by selecting the SHAPE operation from the window operation menu. Reshaping a window means specifying a new region for the window pane. When the right mouse button is released, the window may be reshaped using either the leftmost or middle mouse buttons.

Using the left mouse button, A window anywhere on the screen. First, the cursor must be positioned on the screen where the window is to be located. Then, the left mouse button must be pressed and the mouse dragged to the right and down. A ghost frame will expand indicating the dimensions of the window. When the user is satisfied with the window's shape, release the left mouse button. The contents of the window will be redisplayed in the pane at its new location.

Using the middle mouse button, the current dimensions of the window may be adjusted at its present location. When the middle mouse button is pressed, the cursor leaps to the nearest corner of the window. The mouse may be dragged to adjust the dimensions of the window. Typically, this capability will be used to make small adjustments in the dimensions of a window which is already correctly positioned on the screen.

5.3.6 REDISPLAYING A WINDOW

The contents of a window may be redisplayed by selecting the **REDISPLAY** operation from the window operation menu. Redisplaying may be required for a number of reasons:

1. The contents of the window may have been manipulated using the mouse such that detritus remains in the window.
2. As a result of reshaping the window, the contents have been dislocated within the window.

When the mouse button is released, the contents of the window pane are redisplayed.

5.3.7 PAINTING IN A WINDOW

A user may "paint" within a window when the PAINT operation is selected from the window operation menu. When the rightmost button is released, cursor control switches to a mode where the cursor affects individual bits in the window pane as its traverses them. The three mouse buttons have the effect on the cursor described in Table 5-2.

Table 5-2. Effect of Mouse Buttons During Painting

Mouse Button	Effect
LEFT	When the left mouse button is pressed, bits will be added to the window pane, i.e., each bit traversed by the cursor will be "turned on" (even if it is already on).
MIDDLE	When the middle mouse button is pressed, bits will be erased from the window pane, i.e., each bit traversed by the cursor will be "turned off" (even if it is already off).

RIGHT	Pops up a command menu that allows you to define the characteristics of the "paint brush" or exit painting mode.

The effects of the left and middle mouse buttons are depicted in Figure 5-10.

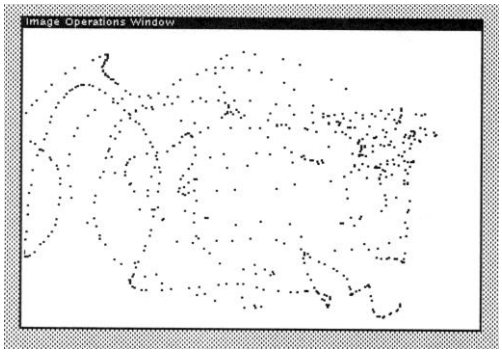

Figure 5-10. Effect of Left Mouse Button During Painting

5.3.7.1 Paint Commands

When the right mouse button is pressed while painting, a command menu appears as depicted in Figure 5-11:

Figure 5-11. Paint Command Menu

Selecting QUIT from the Paint Command Menu leaves the characteristics of the brush unchanged. The other commands are discussed in the following sections.

5.3.7.2 SetMode Command

The *SetMode* command determines how bits painted on the window will interact with bits already appearing in the window. There are three modes which are displayed in an auxiliary pop-up menuas described in Table 5-3.

Table 5-3. New Bits Interaction with Painted Bits

Mode Command	Usage
REPLACE	Bits painted in the window replace existing bits
INVERT	Bits over which the cursor passes are inverted
ADD	Bits over which the cursor passes are added to existing bits

The default operation is to REPLACE existing bits.

5.3.7.3 SetShade Command

The *SetShade* command allows you to set the shade with which bits will be painted on the window. The shades are chosen from a pop-up menu as depicted in Figure 5-12.

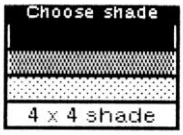

Figure 5-12. SetShade Option Menu

If you choose the 4x4 shade option, you are presented with a pop-up window that allows you to customize the shade which the cursor will assume. This pop-up window is depicted in Figure 5-13. You must use the left mouse button to select squares to darken the shade. You must use the left mouse button to exit from this pop-up window.

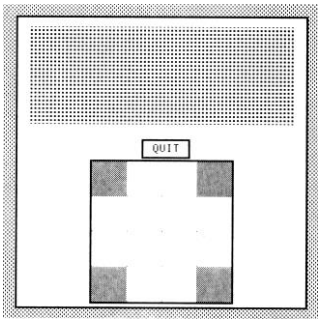

Figure 5-13. 4x4 Shade Customization Window

5.3.7.4 SetShape Command

The *SetShape* command allows you to select the shape of the brush with which you will paint bits on the window. The brush shape is selected from a pop-up menu. The five brush types that are supported are:

1. Diagonal
2. Vertical
3. Horizontal
4. Square
5. Round

Figure 5-14 provides some examples of brush types.

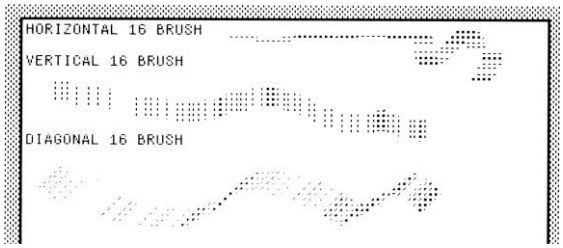

Figure 5-14. Examples of Several Brush Types

5.3.7.5 SetSize Command

The *SetSize* command allows you to determine the size of the brush with which you paint bits on the window. The size is selected from a pop-up menu whose options are: 1, 2, 4, 8, or 16.

5.3.8 TAKING A SNAPSHOT

A user may take a snapshot of a window's contents by selecting the SNAP operation from the window operation menu. When the rightmost mouse button is released, you will be prompted to define a new region on the screen. Using the leftmost mouse button, define the region for a new window. A new window is created of the same dimensions as the existing window. A copy of the contents of the current window is placed in the new window. This operation is useful for saving images in a window for later usage. Figure 515 depicts this operation.

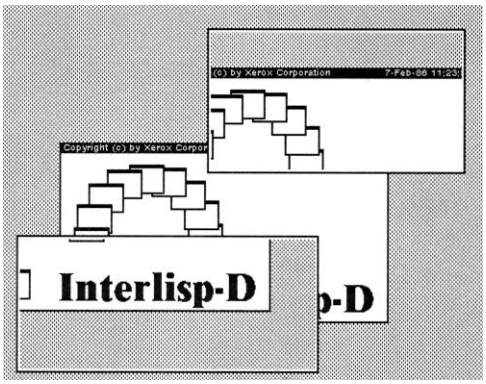

Figure 5-15. Example of a Snapshot of a Window

5.3.9 SHRINKING A WINDOW

Screen real estate is limited. Occasionally, you will find that you create a large number of windows containing useful information. However, this makes is difficult to access underlying windows because you must bring them to the top of the stack. If they are obscured, you must move windows around in order to locate the correct ones to be accessed.

Interlisp allows you to create *icons* to represent windows which, while open, are not fully displayed on the screen. Icons allow you to manage you screen space more efficiently. An icon is a small rectangle containing either text or a bitmap. If text is displayed, it is usually the title of the window. If a bitmap is displayed, it is some iconic representation that is a mnemonic for the windows contents. For example, shrinking the FileBrowser Window displays the icon as shown in Figure 5-16.

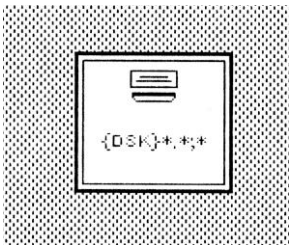

Figure 5-16. FileBrowser Icon from SHRINK command

A window may be shrunk at any time by selecting the SHRINK operation from the window operation menu. When the right mouse button is released, the window disappears from the screen and is replaced by the appropriate icon. You may determine the type of icon that is produced by associating a function with the proper window property.

5.3.10 EXPANDING A WINDOW

When a window is represented by an icon, its contents cannot be manipulated or printed. To access its contents, the window must be expanded so that its pane is visible again. To expand a window, the user must select the EXPAND operation from the window operation menu. When the rightmost mouse button is released, the icon upon which the cursor rests will be expanded into a fully visible window which will be placed at the top of the occlusion stack.

Note that the window operation menu has a slightly different format. The REDISPLAY and CLEAR operations have been removed, and the SHRINK operation has been replaced by the EXPAND operation.

5.3.11 DEFAULT WINDOW OPERATION MENU

The actions associated with the right mouse button may be redefined through the programmable window operations that are discussed in Section 5.5. However, the interactive window operations remain available to the user. Interlisp observes the convention that the interactive window operations will always be available through the title or border of a window. To access the standard window operations menu, place the cursor in the title area or on the border of the window and press the rightmost mouse button.

The default window menu associated with the right mouse button is depicted in Figure 5.8. The window handle for this menu is stored in the system variable *WindowMenu* and its command list is the value of the system variable *WindowMenuCommands*. Their values are shown below:

```
<-WindowMenu
{MENU}#64,101410

<-WindowMenuCommands
((Close         (QUOTE CLOSEW)
      "Closes a Window")
 (Snap          (QUOTE SNAPW)
      "Saves a snapshot of aregion of the screen.")
 (Paint(QUOTE PAINTW)
      "Starts a painting mode in which the mouse can be used to draw pictures or make notes on windows.")
 (Clear         (QUOTE CLEARW)
      "Clears a window to its gray.")
 (Bury          (QUOTE BURYW)
      "Puts a window to the bottom.")
 (Redisplay     (QUOTE REDISPLAYW)
      "Redisplays a window using its REPAINTFN.")
 (Hardcopy      (QUOTE HARDCOPYIMAGEW)
```

```
    "Prints a window using its HARDCOPYFN."
    (SUBITEMS
       ("To a file" (QUOTE HARDCOPYIMAGEW.TOFILE)
       "Puts image on a file; prompts for filename and format")
       ("To a printer" (QUOTE HARDCOPYIMAGEW.TOPRINTER)
       "Sends image to a printer of your choosing")
    ))
(Move       (QUOTE MOVEW)
    "Moves a window by a corner.")
(Shape      (QUOTE SHAPEW)
    "Gets a new region for a window. Left button down marks fixed corner; sweep to other corner.
    Middle button down moves closest corner.")
(Shrink     (QUOTE SHRINKW)
    "Replaces this window with its icon (or title if it doesn't have an icon."))
```

The text that is associated with each function will be displayed in the prompt window if you hold the mouse button down for an appropriate period of time.

5.3.12 BACKGROUND OPERATIONS

When the cursor resides in the background, e.g., it is not in any open window, the right button causes the Background Display Menu to pop-up. This menu is depicted in Figure 5-17.

Figure 5-17. Background Display Menu

5.4 SCROLLING

In many applications the contents of a window may be too large to display within the physical dimensions of the window. This usually occurs when a lengthy text file or a large graphical display is being viewed. Interlisp supports the notion of scrolling within a window. Scrolling allows the user to treat the window pane as a frame which views only a portion of a larger scene behind the window. By moving the frame, the user can view different portions of the screen. You may scroll both vertically and horizontally.

Each object in a window has its own coordinate system. The object may have many components which are related to each other and are laid according to the object's coordinate system (as specified by the display or image stream that was used to print the object in the window). When a window is created, the X-OFFSET and Y-OFFSET of its display stream map the object's origin into the lower left corner of the window's display pane. The clipping region is set to the interior region of the window.

There are several "regions" that are associated with a window as described in Table 5-4.

Table 5-4. Regions of a Window

Region	Description

Object Extent	This region in the window's coordinate system which contains the complete image of the object. It is stored as the value of the EXTENT window property
Clipping Region	This region of the display stream (which is obtained via DSPCLIPPINGREGION) specifies the portion of the object that is actually visible in the window. The clipping region is set so that it corresponds to the interior region of the window.
Window Region	This region specifies the area on the screen that the entire window occupies when it is fully visible. This region is stored as the value of the window's REGION property.

When scrolling is enabled and the user attemps to print lines of text in the window that would run off the bottom, the contents of the window "scroll up" so that the new lines of text become visible. This feature is controlled by DSPSCROLL.

When scrolling is enabled, gray-shaded scrolling bars will appear on the left and bottom edges of the window. The mouse keys are used control the scrolling within the window as described in Table 5-5.

Table 5-5. Controlling Scrolling by Mouse Keys

Mouse Key	Effect
LEFT	This key is used to indicate upward or leftward scrolling by the amount necessary to move the selected position to the top of the window or to its left edge. Thus, if you place the cursor in a position in the scroll bar corresponding to some object in the window, pressing the left mouse button will cause that object to be moved to the top of the window.
MIDDLE	This key is used to indicate a global placement of some object in the window. By placing the cursor in the gray area of the scroll bar, which indicates the amount and portion of the object being viewed, you can manipulate which portion is viewed by pressing the middle mouse button and moving the cursor up or down (or left or right).
RIGHT	This key is used to indicate downward or rightward scrolling by the amount necessary to move the selected position to the top of the window or to its left edge. Thus, if you place the cursor in a position in the scroll bar corresponding to some object in the window, pressing the left mouse button will cause that object to be moved to the top of the window.

When the mouse button is released in a scroll bar, the function SCROLLW is called. SCROLLW calls the function associated with the SCROLLFN property of the window. This function should do the actual scrolling of the window's contents.

5.4.1 SCROLLING A WINDOW

You may scroll the contents of a window using the function SCROLLW:

Function:	SCROLLW
# Arguments:	4
Arguments:	1) WINDOW, a window handle
	2) DELTAX, amount to scroll in the X-direction
	3) DELTAY, amount to scroll in the Y-direction
	4) CONTINUOUSFLG, a flag for continuous scrolling
Value:	The window handle.

This function merely invokes the function, if any, associated with the SCROLLFN property of the window.

5.4.2 HANDLING THE MOUSE DURING SCROLLING

When the cursor is positioned in the scroll bar, you need to track the mouse carefully in order to ensure that the scrolling appears to be continuous. The function **SCROLL.HANDLER** below, allows you to track the cursor:

Function:	SCROLL.HANDLER
# Arguments:	1
Arguments:	1) WINDOW, a window handle
Value:	The window handle.

When the cursor leaves the window in the left or downward direction, SCROLL.HANDLER is called by the mouse handle. If the WINDOW does not have a scroll bar associated with it in this direction, a scroll bar is created and attached to the window that is SCROLLBARIWDTH pixels wide.

SCROLL.HANDLER then waits for SCROLLWAITTIME milliseconds. If the cursor is still located in the scroll bar, it opens a window the size of the scrolling region and changes the cursor to indicate that scrolling is taking place.

When a mouse button is pushed, the cursor shape is changed to indicate the type of scrolling.

If the mouse button is held for WAITBEFORESCROLLTIME milliseconds, the function SCROLLW will be called each WAITBEFORESCROLLTIME milliseconds until the mouse button is released. During these calls, CONTINUOUSFLG is set to T. If the mouse button is released before some interval of WAITBEFORESCROLLTIME milliseconds, then CONTINUOUSFLG is set to NIL. The arguments passed to SCROLLW depend on the mouse button as described in Table 5-6.

Table 5-6. SCROLLW Arguments

Mouse Button	Description
LEFT	In the vertical scroll region (left side of window), DELTAY is the distance from the cursor's position at the time the button was released to the top of the window; DELTAX is 0.
LEFT	In the horizontal scroll region (bottom of the window), DELTAX is the distance from the cursor's position to the left edge of the window; DELTAY is 0.
RIGHT	In the vertical scroll region, DELTAY is the distance from the cursor's position at the time the button was released to the bottom of the window; DELTAX is 0.
RIGHT	In the vertical scroll region, DELTAY is the distance from the cursor's position at the time the button was released to the bottom of the window; DELTAX is 0
MIDDLE	In the horizontal scroll region, DELTAX is the distance from the cursor's position to the right edge of the window; DELTAY is 0

If the window does not have a SCROLLFN window property or its value is NIL, then the window is not scrollable and no scroll regions will be displayed when the cursor exist the window.

5.4.3 SCROLLING BY REPAINTING

The standard scrolling function provided with Interlisp is **SCROLLBYREPAINTFN**. The IRM [IRM 28.4.9] notes that it should be used for most scrolling windows.

Function:	SCROLLBYREPAINTFN
# Arguments:	4
Arguments:	1) WINDOW, a window handle 2) DELTAX, amount to scroll in X-direction 3) DELTAY, amount to scroll in Y-direction 4) CONTINUOUSFLG, a flag for continuous scrolling.
Value:	NIL.

Normally, you would assign this function as the value of the window property SCROLLFN.

When SCROLLBYREPAINTFN is called, it bitblts the bits that are to remain visible after scrolling to their new location in the window. It fills the area vacated by these bits with the background texture, adjusts the window's coordinate system (relative to the display stream), and calls the window's REPAINTFN on the exposed region. This function will scroll any window which has a repaint function. Consider the following functions which create a scrolling window for displaying an expression:

```
(DEFINEQ (CREATE.SCROLLING.WINDOW (EXPRESSION)
(PROG (W1)
    (SETQ W1 (CREATEW NIL "Scrolling Window"))
    (WINDOWPROP W1 'EXPRESSION EXPRESSION)
    (WINDOWPROP W1 'REPAINTFN
            (FUNCTION REPAINT.SCROLL.WINDOW))
    (WINDOWPROP W1 'RESHAPEFN
            (FUNCTION RESHAPE.SCROLL.WINDOW))
    (WINDOWPROP W1 'SCROLLFN
            (FUNCTION SCROLLBYREPAINTFN))
    (RESHAPE.SCROLL.WINDOW W1)
    (RETURN W1))
    ))
```

This function sets up some of the window properties that enable scrolling to be performed. Note that the expression to be displayed is cached in the window on the window's property list.

```
(DEFINEQ (RESHAPE.SCROLL.WINDOW (WINDOW)
      (PROG (BOTTOM REGION)
      (DSPRESET WINDOW)
      (WINDOWPROP WINDOW 'X-ORIGIN (DSPXPOSITION NIL WINDOW))
      (WINDOWPROP WINDOW 'Y-ORIGIN (DSPYPOSITION NIL WINDOW))
      (REPAINT.SCROLL.WINDOW WINDOW)
      (SETQ REGION
          (create  REGION
                LEFT <- 0
                BOTTOM <- (SETQ          BOTTOM
                                          (IPLUS (DSPYPOSITION NIL WINDOW)
                                                 (FONTPROP WINDOW 'ASCENT)))
                WIDTH <- (WINDOWPROP WINDOW 'WIDTH)
                HEIGHT <- (IDIFFERENCE  (WINDOWPROP   WINDOW  'HEIGHT) BOTTOM)))
      (WINDOWPROP WINDOW 'EXTENT REGION))
   ))
```

This function resets the windows X- and Y-coordinates and then repaints the window.

```
(DEFINEQ  (REPAINT.SCROLL.WINDOW (WINDOW REGION)
    (MOVETO     (WINDOWPROP WINDOW 'X-ORIGIN)
                (WINDOWPROP WINDOW 'Y-ORIGIN)
                WINDOW)
    (PRINTDEF   (WINDOWPROP WINDOW 'EXPRESSION)
                0
                NIL
                NIL
                NIL
                WINDOW)
    ))
```

This function moves the cursor to the X- and Y-coordinates of the window's origin and redisplays the expression. These functions were modeled after examples that appeared in the IRM. Note that if the WINDOW has an EXTENT property, the scrolling will be limited in the X and Y directions by the value of the window property SCROLLEXTENTUSE.

If DELTAX or DELTAY is a floating point number, then SCROLLBYREPAINTFN repositions the window contents proportional to the distance from the top and left corner as a proportion of the region given as the EXTENT.

5.4.4 SCROLLING PROPERTIES

Several window properties are used to control scrolling activities as described in the following sections.

5.4.4.1 The Extent of the Window

The EXTENT of a window is a region that is used to limit the scrolling performed by the SCROLLFN. The value of EXTENT is a region in the window's display stream which contains the complete image of the object being viewed by the window. Note that the extent may be smaller than the window region, but that this is atypical. Usually, the extent will not be known in one or both dimensions. This can be indicated to the window operations by specifying a value of -1 for the WIDTH or HEIGHT of the region which is the value of EXTENT.

5.4.4.2 The Scrolling Function

The scrolling function for a window is specified as the value of the window property SCROLLFN. If this value is NIL, the window is not scrollable. The function assigned to this property takes four arguments:

1. The window being scrolled.
2. The distance to scroll in the horizontal direction.
3. The distance to scroll in the vertical direction.
4. A flag which is T if a mouse button is held down while in the scrolling region.

For arguments (2) and (3), a positive number indicates either right or up, while a negative number indicates either left or down.

5.4.4.3 No Scroll Bars

If the window property NOSCROLLBARS is non-NIL, then no scrollbars will be displayed for the window. In addition, mouse-driven scrolling is disabled, but you may still scroll the window under program control using SCROLLW.

5.4.4.4 Scroll Extent

The window property SCROLLEXTENTUSE is used by SCROLLBYREPAINTFN to limit the distance that will be scrolled in the X- and Y-directions. Its possible values are described in Table 5-7.

Table 5-7. Scrolling Extent Property Values

Value	Usage
NIL	The extent is kept visible. The top of the extent region will not be displayed below the top of the window pane. The left of the extent region will not be displayed to the right of the left edge of the window pane
T	The value of EXTENT is not used to control scrolling.
LIMIT	The extent region is always kept visible. The window is only allowed to view within the extent. If the extent region is larger than the current window region, the window is reshaped to cover the extent region (with the limits of the screen).
+	The extent region is kept visible in the positive direction.
-	The extent region is kept visible in the negative direction.
+- or -+	The extent region is kept visible in the window.

(<x-behavior> . <y-behavior>)	The CAR specifies the scrolling limit in the X-direction and the CDR specifies the scrolling limit in the Y-direction. The elements should be one of the atoms NIL, T, LIMIT, +, -, +-, or -+. In this case, NIL is treated the same as LIMIT.

For unlimited scrolling in the Y-direction, you should use the specification (LIMIT . +) for SCROLLEXTENTUSE.

5.5 WINDOW MANAGEMENT FUNCTIONS

The window operations discussed in Section 5.3 are implemented by a set of window management functions which are callable from your program. These functions allow you to perform all of the window operations (plus a few more) under program control.

5.5.1 CREATING A WINDOW

A window may be created from within a program by executing the function **CREATEW**.

Function:	CREATEW
# Arguments:	4
Arguments:	1) REGION, a specification of the window size
	2) TITLE, a string naming the window
	3) BORDER, the width of the border in bits
	4) NOOPENFLAG, to open or not
Value:	A window handle.

CREATEW creates a new window as a data structure in memory and returns the handle of the window object as its value.

REGION specifies the left and bottom coordinates of the window on the display screen and the height and width of the window. The usable height and width are less than the maximum height and width specified by the region parameters (see below). If REGION is NIL, GETREGION is called to prompt you to interactively specify a region with the mouse.

TITLE is a string that specifies the name of the window. The title is displayed in the top border using the global display stream *WindowTitleDisplayStream*. The height of the title is determined by the font currently associated with that display stream. The default height may be determined by FONTPROP.

```
<-(FONTPROP WindowTitleDisplayStream 'HEIGHT)
9
```

BORDER specifies the size (in bits) of the border outlining the window. The default border size, given by *WBorder*, is 4 bits. If BORDER is NIL, the default border size will be used.

NOOPENFLG is a flag that determines whether or not the window will be opened (e.g., displayed on the screen) when it is created. If NOOPENFLG is non-NIL, the window will not be opened. This flag is useful when you are initializing an application because it allows you to create all of the windows required, but open them only when they are actually needed.
Figure 5-18 depicts the effect of CREATEW.

```
<-newWindow <-(CREATEW (CREATEREGION 100 100 200 200) “Example Window”) 3 NIL)
{WINDOW}#64,77640
```

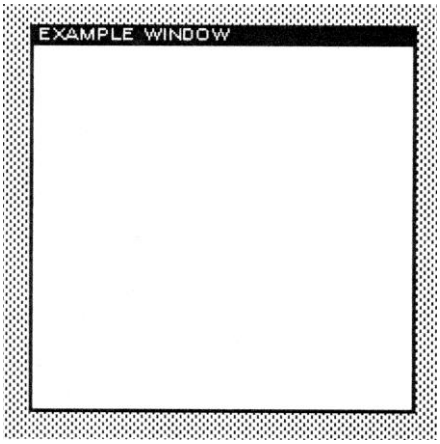

Figure 5-18. CREATEW Example

Note that the new window is empty and displays the background shade. Usually, you will create a new window with the NoOPENFLG set to T, then populate the window before opening it for the first time using OPENW.

Note that when you create a window, you do not specify its origin on the screen. The origin of the window is specified by the values of LEFT and BOTTOM of the region.

5.5.1.1 Usable Area

The usable area of a window is reduced by the size of the borders. The region parameters always specify the maximum size of the window. The usable size is determined by subtracting the border sizes from the dimensions given by the region parameters.

If the default border size is used, then the usable height of the window is reduced by (2 x WBorder + Title Size) and the usable width is reduced by (2 x WBorder). Otherwise, the usable height and width are determined by substituting the value of BORDER for WBorder.

Consider the following example. Let us create a window whose region is 100 by 200. The maximumsize of the window is 100 bits by 200 bits. Using the defaults, the usable size of the window is calculated as follows:

```
<-WBorder
4
<-(FONTPROP WindowTitleDisplayStream 'HEIGHT)
9
```

So the usable height is 83 bits and the usable width is 192 bits.

Let us define functions which compute the usable height and width of a window:

```
<-(DEFINEQ (usable.width (window)
      (DIFFERENCE    (WINDOWPROP window 'WIDTH)
                     (COND
                           ((WINDOWPROP window 'BORDER)
                            (ITIMES 2 (WINDOWPROP window 'BORDER)))
                           (T (ITIMES 2 WBorder))
                     ))
   ))
(USABLE.WIDTH)
```

```
<-(DEFINEQ (usable.height (window)
        (DIFFERENCE     (WINDOWPROP window 'HEIGHT)
                        (IPLUS      (FONTPROP WindowTitleDisplay Stream 'HEIGHT)
                                    (COND
                                            ((WINDOWPROP window 'BORDER)
                                             (ITIMES 2 (WINDOWPROP window 'BORDER)))
                                            (T  (ITIMES 2 WBorder)))))
    ))
(USABLE.HEIGHT)
```

Consider the following examples of their usage:

```
<-(SETQ XW (CREATEW (CREATEREGION 49 31 401 308)))
{WINDOW}#64,77640

<-(USABLE.WIDTH XW)
385

<-(USABLE.HEIGHT XW)
283
```

The user will often need to know the usable height and width of a window when calculating the value of EXTENT for scrolling.

5.5.1.2 A Window Example

Let us create a window on the display screen. The window will be 200 x 200 bits in extent with the default border. Its title will be "Example Window". The window will be placed on the display screen so that its lower lefthand corner is positioned at absolute coordinates (100, 100). To create this window, we execute the following expressions:

```
<-(SETQ aregion (CREATEREGION 100 100 200 200))
(100 100 200 200)

<-(SETQ awindow (CREATEW aregion "Example Window" 3 NIL)
{WINDOW}#66,2234
```

And the window appears on the display screen as depicted in Figure 5-18.

The structure of the object representing the window may be viewed by the Inspector. It contains the following properties and values:

```
SCREEN                  NIL
WINDOWENTRYFN           GIVE.TTY.PROCESS
PROCESS                 NIL
WBORDER                 4
NEWREGION               NIL
WTITLE                  "Example Window"
MOVEFN                  NIL
CLOSEFN                 NIL
HORIZSCROLLWINDOW       NIL
VERTSCROLLWINDOW        NIL
SCROLLFN                NIL
HORIZSCROLLREG          NIL
VERTSCROLLREG           NIL
USERDATA                NIL
EXTENT                  NIL
RESHAPEFN               NIL
REPAINTFN               NIL
CURSORMOVEDFN           NIL
```

CURSOROUTFN	NIL
CURSORINFN	NIL
RIGHTBUTTONFN	NIL
BUTTONEVENTFN	TOTOPW
REG	(100 100 200 200)
SAVE	{BITMAP}#74,124322
NEXTW	{WINDOW}#74,25150
DSP	{STREAM}#74,125234

NEXTW is a pointer to the next window in the list of active windows.

The rest of these properties will be discussed in Section 5.6.

5.5.1.3 Decoding Window Arguments

Another function for creating a window is **DECODE.WINDOW.ARG**.

Function:	DECODE.WINDOW.ARG
# Arguments:	6
Arguments:	1) WHERESPEC, the location of the window 2) WIDTH, the window width 3) HEIGHT, the window height 4) TITLE, the window title 5) BORDER, the window border 6) NOOPENFLG, whether or not to open the window
Value:	A window handle.

DECODE.WINDOW.ARG examines its various arguments in different combinations before passing them to CREATEW. WHERESPEC may be one of the following:

- A region which is adjusted to be on the screen.
- A position whence it specifies the lower left corner of the window.
- A window which is returned immediately without calling CREATEW.
- NIL, whence the user is prompted to specify the region.

A window is open when it is displayed on the display screen. Otherwise, it is closed. Thus, a window may exist and not be open. The existence of a window is indicated by the existence of a window handle which is assigned as the value of a variable. Windows may be opened and closed at any time.

5.5.2 OPENING A WINDOW

To open a window that is currently closed, you execute the function **OPENW** which takes the form:

Function:	OPENW
# Arguments:	1
Argument:	1) WINDOW/STREAM, a window or stream handle.
Value:	The window or stream handle.

If WINDOW/STREAM is a closed window or the stream associated with a window, OPENW calls any functions that are specified as the value of the window property OPENFN. There are three cases as described in Table 5-8.

Table 5-8. OPENFN Values

Value	Usage
A non-null list of functions	The functions are called in sequence to display the window (and its contents, if any) on the screen.
One of the values of OPENFN is DON'T or one of the functions returns DON'T	The window will not be displayed on the screen. Typically, this is the only value of OPENFN when this type of behavior is desired.

NIL	The window is placed on the occlusion stack and displayed on the screen. Its location is determined by its current region parameters.

Note that when a window is opened, it is always brought to the top of the occlusion stack. If the window or stream is already open, OPENW returns NIL. Consider the following example:

```
<-(SETQ awindow (CREATEW (CREATEREGION 200 200 300 300)))
{WINDOW}#65,173234

<-(OPENW awindow)
NIL
```

because AWINDOW was opened by the call to CREATEW.

A window may also be opened by displaying something in it. Printing to a closed window implies that you want to see the contents of the message. For example,

```
<-(CLOSEW awindow)
CLOSED

<-(PRINT "Hi There" awindow)
"Hi There"
```

which opens AWINDOW and displays the phrase "Hi There" in the upper left corner.

5.5.3 CLOSING A WINDOW

An open window maybe closed by executing **CLOSEW**.

Function:	CLOSEW
# Arguments:	1
Argument:	1) WINDOW/STREAM, a window or stream handle
Value:	The atom CLOSED.

If WINDOW/STREAM is an open window or the stream associated with an open window, CLOSEW calls any functions which are assigned as the value of the window property CLOSEFN. There are three cases as described in Table 5-9.

Table 5-9. CLOSEFN Values

Value	Usage
A non-null list of functions	The functions are called in sequence to close the window. This allows preprocessing of the window's contents before closing the window (such as saving certain bit patterns)
One of the values is DON'T or one of the functions returns the value DON'T	The window will not be closed and CLOSEW will return NIL
NIL	The window is removed from the occlusion stack. The bits that it obscured are redisplayed on the screen.

If the window or stream is already closed, NIL is returned by CLOSEW. Consider the following example:

```
<-(SETQ awindow (CREATEW (CREATEREGION 200 200 300 300)))
{WINDOW}#65,173234

<-(CLOSEW awindow)
CLOSED
```

```
<-(CLOSEW awindow)
NIL

<-awindow
{WINDOW}#65,173234
```

NOTE: Closing a window does not delete the window handle associate with the window. By sending output to the display stream associated with the window, you cause it to be displayed on the screen.

5.5.4 REMOVING A WINDOW

A window may be closed and its window handle released using the function **REMOVEWINDOW**:

Function:	REMOVEWINDOW
# Arguments:	1
Arguments:	1) WINDOW, a window handle
Value:	NIL.

Note that closing a window does not release its window handle or the storage associated with a window. REMOVEWINDOW closes a window if it is open and releases the window handle. Attempting to reopen the window will not succeed because the window handle has been released. Consider the following example:

```
<-(OPENW awindow)
(WINDOW}#66,2234

<-(REMOVEWINDOW awindow)
NIL
```

Once a window's handle has been released, it cannot be opened.

5.5.5 TESTING WINDOWS

The status of a window of may be tested, its identity, and whether it is open or not. The user may also test whether or not the Window Manager has been enabled.

5.5.5.1 Testing for Window Existence

The user may determine whether an arbitrary Lisp object is a window handle or not by executing **WINDOWP**:

Function:	WINDOWP
# Arguments:	1
Argument:	1) X, an arbitrary Lisp object
Value:	The value of X if it is a window; otherwise, NIL.

WINDOWP determines if the value of X is a window handle. If so, it returns that value; otherwise, NIL. Consider the following examples:

```
<-(SETQ awindow (CREATEW aregion "Example Window" NIL 3))
{WINDOW}#74,25000

<-(WINDOWP awindow)
{WINDOW}#74,25000
```

5.5.3.2 Testing for an Open Window

The user may test whether or not a window is open by executing **OPENWP**:

Function:	OPENWP
# Arguments:	1
Argument:	1) WINDOW, a window handle
Value:	The window handle if WINDOW represents an open window.

OPENWP returns the window handle if the window it represents is an open window; otherwise, NIL. Let AWINDOW contain the address of a window handle. Then, we can test whether or not the window is open using the following expression:

```
<-(OPENWP awindow)
{WINDOW}#74,25000
```

5.5.3.3 Obtaining the Open Windows

The user may obtain a list of all open windows by executing **OPENWINDOWS**:

Function:	OPENWINDOWS
# Arguments:	0
Arguments:	N/A
Value:	A list of the handles of open windows.

OPENWINDOWS returns the value of the occlusion stack. When Interlisp is initialized, three windows appear on the screen. Executing OPENWINDOWS, we obtain the list:

```
<-(OPENWINDOWS)
({WINDOW}#74,25554 {WINDOW}#74,25640 {WINDOW}#74,25470)
```

Of course, the addresses of the window handles depend on how much memory you have in your machine, whether or not you do any automatic processing at logon (such as file loading), and the condition of your SYSOUT.

It is difficult to correlate the window handles with the windows on the screen. You can determine which handles refer to the prompt window, the logo window, and the executive window fairly easily. However, for windows opened by your application, it is useful to keep their handles in separate system variables.

5.5.3.4 Determining the Window of a Position

In many cases, the user will want to know which window contains a particular position as indicated by a pair of X-Y coordinates. This is useful when you point at something on the screen with the mouse and expect a program to respond. The program must determine the response based on the current window and its associated properties. Which window contains a position may be determined by executing **WHICHW**.

Function:	WHICHW
# Arguments:	2
Arguments:	1) X, an X-axis coordinate
	2) Y, a Y-axis coordinate
Value:	The window handle containing the position.

X may be a position, whence the value of Y will be ignored. If X and Y are coordinates, they must both be numbers valid within the screen's coordinate system. Let AWINDOW contain the window handle of the window whose region is (100 100 200 200). We can test the point (150 150) as follows:

```
<-(WHICHW 150 150)
{WINDOW}#74,25000
```

If X and Y are NIL, then WHICHW uses the current position of the cursor:

```
<-(WHICHW)
```

{WINDOW}#74,25000

after I have placed the cursor in the window AWINDOW. And, moving the cursor to the Logo window:

```
<-(WHICHW)
{WINDOW}#74,25554
```

The position given by X,Y may occur in more than one window if a number of windows are stacked on top of one another. When this occurs, WHICHW returns the window handle of the topmost window.

5.5.3.5 Determining if Windows Are Enabled

The user may determine (usually from within a program) whether or not the Window Manager is enabled using the function **WINDOWWORLDP**.

Function:	WINDOWWORLDP
# Arguments:	0
Arguments:	N/A
Value:	T, if the Window Manager is enabled; otherwise, NIL.

This function is meant to be used from within a program which might be ported between different environments (e.g., Interlisp-10 and Interlisp). Rarely will you want to disable the Window Manager in the Interlisp environment. At the top level, it returns T as follows:

```
<-(WINDOWWORLDP)
T
```

5.5.3.6 Determining the Active Windows

The active windows that are known to Interlisp may be determined using the function **ACTIVEWINDOWS**.

Function:	ACTIVEWINDOWS
# Arguments:	0
Arguments:	N/A
Value:	A list of window handles.

The list of open windows, which is returned by OPENWINDOWS, is a subset of the list of active windows. Consider the following example:

```
<-(ACTIVEWINDOWS)
({WINDOW}#64,152150 {WINDOW}#65,20470   {WINDOW}#60,164770
 {WINDOW}#65,20064   {WINDOW}#64,152320 {WINDOW}#64,152234
 {WINDOW}#64,152000 {WINDOW}#74,25460   {WINDOW}#74,25470)
```

The set of active windows includes all windows for which window handles exist. However, the windows may not be open whence they will not appear in the result of OPENWINDOWS.

5.5.6 DETERMINING WINDOW ATTRIBUTES

As mentioned above, the behavior of a window is controlled by a set of window properties. Some window properties are already defined by the Interlisp Window Manager. You may define additional window properties for your own use as long as their names do not conflict with those used by the Window Manager. Window properties are retrieved and set using the functions described in the following sections.

5.5.6.1 Retrieving and Setting a Window Property

The user may retrieve or set a window property using the function **WINDOWPROP**.

Function:	WINDOWPROP
# Arguments:	2-3
Arguments:	1) WINDOW, a window handle
	2) PROP, a window property name
	3) NEWVALUE, a value to be stored in PROP
Value:	The previous value of PROP.

WINDOWPROP is a nospread function. It returns the previous (or current) value of the window property. Consider the following example:

```
<-XW
{WINDOW}#57,15320

<-(WINDOWPROP XW 'BORDER)
4
```

If NEWVALUE is given, it is stored as the new value of the window property. Consider the following example:

```
<-(WINDOWPROP XW 'BORDER 16)
4
```

When the window is next redisplayed, its border will be changed. If NEWVALUE is specified as NIL, then NIL is stored as the new value of the window property. Consider the following example:

```
<-(WINDOWPROP XW 'BORDER NIL)
16
```

Note that this differentiates from the case where the third argument is not specified which indicates that the value of PROP is to be retrieved. If PROP is not recognized as a window property, it is stored on a property list under the property USERDATA. Consider the following example:

```
<-(WINDOWPROP XW 'PAINTBRUSH)
(PAINT 36873 (DIAGONAL 16))
```

Note that the property USERDATA cannot be directly accessed by WINDOWPROP.

Some window properties cannot be set by the user because they are intimately tied with the Window Manager's management of the interactive display environment. Attempting to set such properties will cause an error.

5.5.6.2 Adding a New Window Property

The user may add a new item to a window property using the function **WINDOWADDPROP**.

Function:	WINDOWADDPROP
# Arguments:	3
Arguments:	1) WINDOW, a window handle
	2) PROP, a window property name
	3) ITEMTOADD, the value of an item to add to PROP
Value:	NIL.

WINDOWADDPROP adds a new item to the end of the list of items constituting the value of the window property. Consider the following example:

```
<-(WINDOWPROP IOWINDOW 'MOVEFN)
NIL

<-(WINDOWADDPROP IOWINDOW 'MOVEFN (FUNCTION MOVE.IT))
NIL
```

```
<-(WINDOWPROP IOWINDOW 'MOVEFN)
(MOVE.IT)
```

If the item is already present (via EQ) in the list of values, nothing is returned. If the current value of the window property is not a list, it is converted to a list before the new item is added.

Typically, WINDOWADDPROP will be used to add functions associated with one or more window properties such as OPENFN or CLOSEFN to the list of functions which is the value of the property. Also, you may define any properties required for your application which are associated with a window. You should ensure that you do not use names for your properties that are the same as window properties.

5.5.6.3 Deleting an Item from a Window Property

The user may delete an item from a list of items that is the value of a window property using the function **WINDOWDELPROP.**:

Function:	WINDOWDELPROP
# Arguments:	3
Arguments:	1) WINDOW, a window handle
	2) PROP, a window property name
	3) ITEMTODELETE, the value of the item to be deleted from PROP
Value:	The previous value of PROP.

WINDOWDELPROP deletes an item from a list of items which constitute the value of the window property. If ITEMTODELETE was a member of the list, the previous value of the list is returned. Consider the following example:

```
<-(WINDOWPROP IOWINDOW 'MOVEFN)
(MOVE.IT)

<-(WINDOWDELPROP IOWINDOW 'MOVEFN (FUNCTION MOVE.IT))
(MOVE.IT)

<-(WINDOWPROP IOWINDOW 'MOVEFN)
NIL
```

If ITEMTODELETE was not a member of the list, NIL is returned:

```
<-(WINDOWPROP IOWINDOW 'MOVEFN)
NIL

<-(WINDOWDELPROP IOWINDOW 'MOVEFN (FUNCTION MOVE.IT))
NIL
```

5.5.7 SHAPING AND SHRINKING WINDOWS

Windows do not have to remain the same size once you have created them. You may adjust their size to fit the available screen real estate, the importance of the window, or the amount of information it must display.

5.5.7.1 Shaping a Window

When a window is created, it is given certain dimensions. Sometimes, these dimensions are arbitrarily chosen and need to be modified to accomodate more information or a different screen position. You may change the shape of a window by assigning it a new region by executing **SHAPEW**:

Function:	SHAPEW
# Arguments:	2

Arguments:	1) WINDOW, a window handle
	2) NEWREGION, a new region handle
Value:	The window handle.

SHAPEW calls the functions, if any, which are the value of the window property RESHAPEFN. If WINDOW is open, it is reshaped to conform to the parameters given by NEWREGION. If NEWREGION is NIL, GETREGION is called to prompt the user for a region specification using the mouse. Let us define a new region - NEWREGION - via:

```
<-(setq newregion (create region 400 100 200 200))
(400 100 200 200)

<-(SHAPEW awindow newregion)
{WINDOW}#74,25000
```

which erases the previous display of the window on the screen and redisplays it at the location specified by NEWREGION.

5.5.7.2 Shaping a Window to a Region

SHAPEW1 changes a window's size and position on the display screen to be a specified region. It takes the form:

Function:	SHAPEW1
# Arguments:	2
Arguments:	1) WINDOW, a window handle
	2) REGION, a region description
Value:	A new window handle.

After clearing the specified region of the display screen, SHAPEW1 invokes the window's RESHAPEFN with three arguments:

1. the window handle
2. a bitmap containing the window's previous screen image
3. the region of the window's old image within the bitmap

Consider the following example in which we reshape XW to a new region specification:

```
<-(SHAPEW1 XW (CREATEREGION 1 1 300 300))
{WINDOW}#64,77640

<-(WINDOWSIZE XW)
(300 . 300)

<-(WINDOWPOSITION XW)
(1 . 1)
```

5.5.7.3 Shrinking a Window

In many cases, a window is used to display information for short periods of time. It is expensive to open and close a window repeatedly, and much more expensive to create and destroy windows. Interlisp allows you to shrink a window to an icon which represents the window.

Shrinking windows is a method of screen space management. Many applications have a large number of windows occupying the screen although not all of these windows are accessed concurrently. By shrinking windows into icons, you make screen space available for other windows while creating an indicator that a window exists.

An ***icon*** is a small rectangle containing text or bitmap which represents a particular window.

To shrink a window to an icon, the user must execute **SHRINKW**.

Function:	SHRINKW
# Arguments:	4
Arguments:	1) WINDOW, a window handle 2) TOWHAT, the icon specification 3) ICONPOSITION, a position for the icon on the screen 4) EXPANDFN, an icon expansion function
Value:	The window handle for the icon.

When we shrink the window without specifying an icon, the icon becomes a rectangle (in inverted video) enclosing the title of the window. The window handle for the icon is returned:

```
<-(SHRINKW awindow)
{WINDOW}#66,2404
```

When Interlisp shrinks a window to an icon, it records the information necessary to expand the icon to a full window under the property USERDATA. The information stored under this property is:

```
(ICONWINDOW {WINDOW}#65,173640
 ICONPOSITION (400 . 100)
 OPENFN (CLOSEICONWINDOW))
```

CLOSEICONWINDOW is an internal function of the Interlisp kernel.

ICONPOSITION specifies the position that the icon will assume on the display screen. If it is NIL, the icon will be placed at the corner of the window furthest from the center of the screen.

TOWHAT, if given, indicates the image that the icon representing the window will have. It can have one of the values described in Table 5-10.

Table 5-10. TOWHAT Values

Value	Usage
String, Atom, or List	If TOWHAT is a string, atom, or list, the icon will be represented as a title only window with the value of TOWHAT displayed as the title.
Bitmap	If TOWHAT is a bitmap, the icon's image will be a copy of the bitmap. Figure 5-19 depicts the icon representing the File Browser.
Window	If TOWHAT is a window handle, that window will be used as the icon.
NIL	If TOWHAT is NIL, then the following rules are applied to determine how to create an icon for the window: 1. If the window has an ICONFN property, the function which is the value of this property is called with two arguments: the window handle and the previously created icon, if any. 2. If the window has an ICON property, it is used as the value of TOWHAT. 3. If the window has neither the ICONFN or ICON property, the icon will be the window's title, but if the window has no title, it will be merely the date and time that the icon was created.

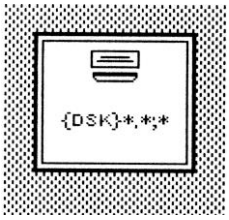

Figure 5-19. Icon Representing the File Browser

5.5.5.4 Expanding an Icon

When a window has been shrunk to an icon, you may not perform any operations upon the window until it has been expanded. To expand an icon to the full window representation, the user must execute **EXPANDW**.

Function:	EXPANDW
# Arguments:	1
Argument:	1) ICON, an icon specification
Value:	The window handle for the window corresponding to the icon.

EXPANDW erases the icon from the screen, opens the window to the region specified in its handle, and redisplays the window's contents. Consider the following example:

```
<-(EXPANDW (WINDOWPROP FBW 'ICONWINDOW))
{WINDOW}#74,25064
```

This opens a window associated with the File Browser. I had previously saved the File Browser's window handle in the variable FBW.

5.5.8 MOVING WINDOWS

In many applications, you will find it advantageous to reorganize the screen real estate from time to time. You may reorganize in two ways:

1. Shrink some windows to icons
2. Move some windows to different positions to make them less obscured

5.5.8.1 Moving a Window to a New Position

A window may be moved to a new position by executing **MOVEW**:

Function:	MOVEW
# Arguments:	3
Arguments:	1) WINDOW, a window handle
	2) POSorX, an X-axis coordinate or a position
	3) Y, a Y-axis coordinate
Value:	The new position of the window;
	otherwise, NIL if the window can't be moved.

If the window is not open when MOVEW is called, the window is moved without being opened. For this operation to occur, POSorX must be non-NIL. Otherwise, the window is opened, because the user must be prompted to specify the location of the new window.

If WINDOW has the atom DON'T as the value of its MOVEFN window property, the window will not be moved. If WINDOW has a non-NIL MOVEFN window property, it should be a list of functions that will be called before the window is moved.

If WINDOW is moved and it has an AFTERMOVEFN window property, the value of this property should be a list of functions that will be called after the window is moved.

If MOVEW moves any part of the window from off-screen onto the screen, that part of the window is redisplayed by calling REDISPLAYW with WINDOW as its argument.

MOVEW operates according to a set of rules as described in the following paragraphs.

POSorX is NIL

If POSorX is NIL, the Window Manager calls GETBOXPOSITION to read the position from the user. If WINDOW has a CALCULATEREGION window property, the associated function is called with WINDOW as an argument to calculate the new region for the window. This function should return a region which will be used to prompt the user. If WINDOW does not have such a window property, the region of the window is used to prompt the user. Consider the following example:

```
<-(SETQ awindow (CREATEW (CREATREGION 200 200 300 300)))
{WINDOW}#65,125000

<-(MOVEW awindow)
(409 . 38)
```

which is the new position of the lower left corner.

```
<-(WINDOWPROP awindow 'REGION)
(409 38 300 300)
```

POSorX is a Position

If POSorX is a position, POSorX is used the move the window's lower left corner to the new absolute screen coordinate. Consider the following example:

```
<-(SETQ awindow (CREATEW (CREATEREGION 200 200 300 300)))
{WINDOW}#65,125000

<-(MOVEW awindow (POINT 150 150))
(150 . 150)

<-(WINDOWPROP awindow 'REGION)
(150 150 300 300)
```

POSorX and Y are Both Numbers

If POSorX and Y are both numbers (e.g., they satisfy NUMBERP), the Window Manager creates a position from the two numbers and uses it to specify the new lower left corner coordinates of the window. Consider the following example:

```
<-(SETQ awindow (CREATEW (CREATEREGION 200 200 300 300)))
{WINDOW}#74,25554

<-(MOVEW awindow 150 150)
(150 . 150)

<-(WINDOWPROP awindow 'REGION)
(150 150 300 300)
```

POSorX is a Region

If POSorX is a region, a position is created by selecting its LEFT as the X-coordinate of a position and its BOTTOM as the Y-coordinate of a position which will specify the new lower left corner of the window. Consider the following example:

```
<-(SETQ awindow (CREATEW (CREATEREGION 200 200 300 300)))
{WINDOW}#74,25554

<-(SETQ aregion (CREATEREGION 150 175 400 400))
(150 175 400 400)

<-(MOVEW awindow aregion)
(150 . 175)

<-(WINDOWPROP awindow 'REGION)
(150 175 400 400)
```

5.5.8.2 Relative Window Displacement

MOVEW moves a window to a location whose position is composed from absolute screen coordinates. **RELMOVEW** moves a window to a position relative to its current location. It takes the form:

Function:	RELMOVEW
# Arguments:	2
Arguments:	1) WINDOW, a window handle
	2) POSITION, a position handle
Value:	NIL.

Consider the following example to move a window left and down one screen point:

```
<-(SETQ XW (CREATEW (CREATEREGION 1 1 300 300)))
{WINDOW}#64,77640
```

which positions the window in the lower left hand corner of the screen, and

```
<-(WINDOWPROP XW 'REGION)
(1 1 300 300)

<-(RELMOVEW XW (POINT -1 -1))
NIL
```

which moves part of the window off the screen.

```
<-(WINDOWPROP XW 'REGION)
(0 0 300 300)

<-(RELMOVEW XW (POINT 100 100))
NIL

<-(WINDOWPROP XW 'REGION)
(100 100 300 300)
```

5.5.8.3 Moving a Window to the Top

When a window is displayed on the screen, it may be obscured in whole or in part by one or more windows. You may move a window to the top of the occlusion stack and, thus, make it completely visible by executing **TOTOPW**:

Function:	TOTOPW
# Arguments:	2
Arguments:	1) WINDOW, a window handle
	2) NOCALLTOTOPFNFLG, a flag
Value:	The window handle.

If WINDOW is closed, it is opened by the Window Manager. However, this is also done whenever you display text or graphics in a partially obscured or closed window.

If NOCALLTOTOPFNFLG is NIL, the function associated with the TOTOPFN window property, if any, is called; otherwise, it is not. This permits the function assigned as the value of TOTOPFN to call TOTOPW without causing an infinite loop.

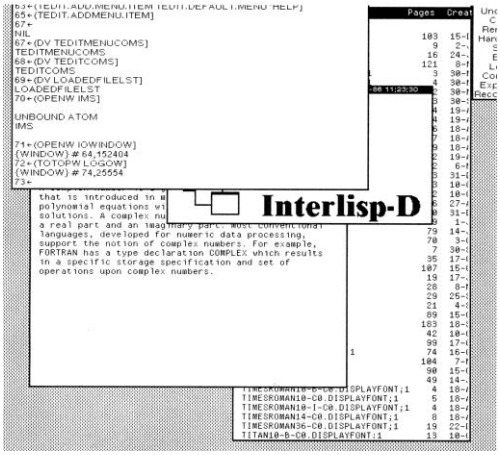

Figure 5-20. Obscured Logo Window

Figure 5.20 depicts three windows in which the Interlisp Logo Window is obscured. Consider the following example:

```
<-(TOTOPW LW)
{WINDOW}#74,25554
```

Figure 5.21 depicts the result of executing TOTOPW.

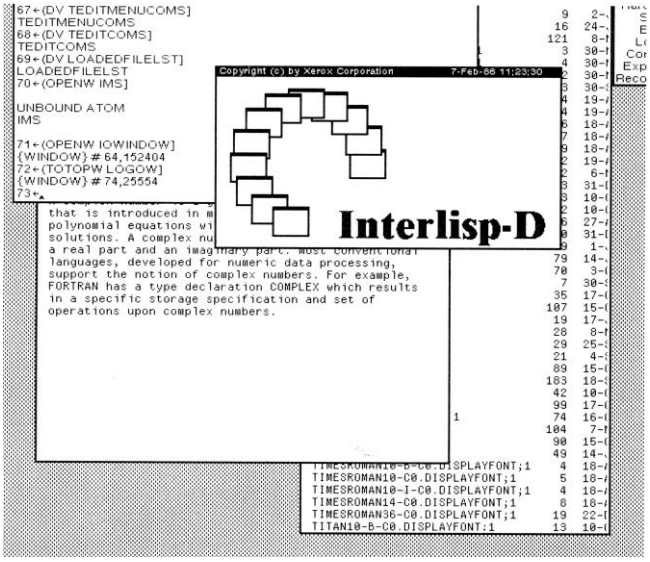

Figure 5-21. TOTOPW Example

5.5.8.4 Burying a Window

When a window is displayed on the screen, it may obscure in whole or in part one or more other windows. To make one or more of those windows more visible, you may bury the obscuring window by moving it to the bottom of the occlusion stack. **BURYW** buries a window, i.e., it makes it least visible by placing it at the bottom of the occlusion stack. It takes the form:

Function:	BURYW
# Arguments:	1
Argument:	1) WINDOW, a window handle
Value:	The window handle.

Consider the following example:

```
<-(BURYW LW)
{WINDOW}#74,25554
```

which returns the Logo Window to the position it previously occupied in Figure 5.20.

5.5.9 CLEARING AND REDISPLAYING WINDOWS

When a window is displayed on the screen and data is written to it, the data is positioned according to the coordinates specified in the function writing to the window. Typically, when text is written to a window, its contents scroll upwards thus maintaining a clean interface. However, when drawing on a window, the contents may obscure each other or overwrite each other. Two functions permit you to refresh the contents of a window: **CLEARW** and **REDISPLAYW**.

5.5.9.1 Clearing a Window

The current contents of a window may be cleared using the function **CLEARW**.

Function:	CLEARW
# Arguments:	1

Arguments:	1) WINDOW, a window object
Value:	NIL

CLEARW erases the current contents of the bit map associated with the window. It then fills the window with its background texture. The X and Y coordinates of the window are set to the left margin and the top of the window less the font ascent. For most windows the background texture will be WHITESHADE. Consider the following example:

```
<-(SETQ awindow (CREATEW aregion))
{WINDOW}#74,25000

<-(PRINT "Hi There" awindow)
"Hi There"

<-(CLEARW awindow)
NIL
```

5.5.9.2 Redisplaying a Window

An entire window or just a region of it may be redisplayed using the function **REDISPLAYW**.

Function:	REDISPLAYW
# Arguments:	3
Arguments:	1) WINDOW, a window object
	2) REGION, a region object
	3) ALWAYSFLG, a flag specifying when the window is to be redisplayed.
Value:	The window handle.

Typically, you will redisplay an entire window whence you will set REGION to NIL. Otherwise, you may redisplay a region of a window by providing the appropriate region handle. Consider a window composed of a number of regions. One region may control the contents of another region such as one region accepting the name of a country and another region displaying its map and key cities. Changing the name of the country in the first region will automatically force the contents of the second region to change.

ALWAYSFLG determines how redisplay requests are handled. If T, the contents of the region of the window are always redisplayed. However, if ALWAYSFLG is NIL, and the window does not have a repaint function, the contents of the window remain unchanged and an error message is displayed in the prompt window. The text of the error message is: "Window has no REPAINTFN. Can't Redisplay".

5.5.10 TESTING FOR A FULL PAGE

A "full page" is the number of characters which, when written to a window, will cause the contents of the window to scroll upwards when the next character is written to the window. You may test for the page full condition using the function **PAGEFULLFN**.

Function:	PAGEFULLFN
# Arguments:	1
Arguments:	1) WINDOW, a window object
Value:	T or NIL.

PAGEFULLFN is invoked if the PAGEFULLFN property of the window is NIL. It is the default function to be invoked when a "page full" condition is detected in a window. PAGEFULLFN returns T if there are characters remaining in the type-in buffer for the window. Otherwise, it inverts the window and waits for you to type a character. When you type a character the contents of the window scroll upwards to accommodate the new character. Consider the following example:

```
<-(PAGEFULLFN XW)
NIL
```

You may demonstrate this behavior by the following example:

```
<-(APROPOS 'TEXT T)
<a big list of atoms and their values>
```

APROPOS prints all atoms (and their values) which have the string "TEXT" in their name to the Executive Window. When APROPOS has filled the window, PAGEFULLFN is called. It inverts the window and waits for you to type something. Note that a caret is placed after the last character typed at the bottom of the window. Figure 5-22 depicts this condition.

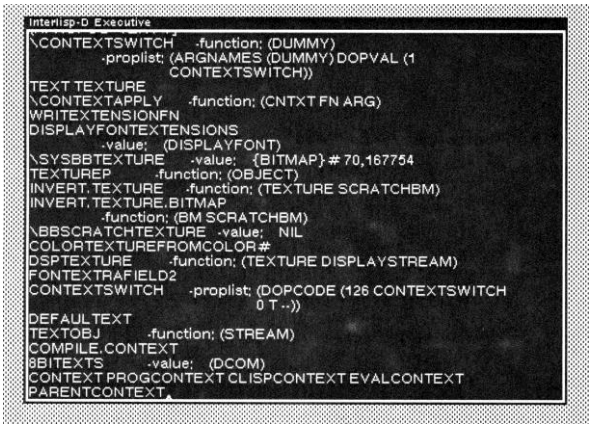

Figure 5-22. Page Full Condition

5.5.11 RESHAPING A WINDOW BY REPAINTING

A window may be reshaped by repainting the contents of the window using the function **RESHAPEBYREPAINTFN** :

Function:	RESHAPEBYREPAINTFN
# Arguments:	4
Arguments:	1) WINDOW, a window handle
	2) OLDIMAGE, a bitmap handle
	3) IMAGEREGION, a region within the bitmap specified by OLDIMAGE
	4) OLDSCREENREGION, the old screen region of the window
Value:	The window handle.

RESHAPEBYREPAINTFN is the default function for the window property REPAINTFN. It bit-blits the contents of the old region, given by IMAGEREGION, into the new region which is determined by executing (WINDOWPROP <window> 'REGION). If the new window shape is larger in either or both dimensions, the newly exposed areas are redisplayed by calling the function associated with the window property REPAINTFN.

WINDOW is the handle of a window that has been reshaped from the screen region (in absolute display system coordinates) to a new shape as determined above. OLDIMAGE is the bitmap handle of the bitmap containing the contents of the window. IMAGEREGION is the region within OLDIMAGE that contains the old image.

RESHAPEBYREPAINTFN determines which areas of the window's contents to remove or extend as follows:

1. If WINDOW's new region shares an edge with OLDSCREENREGION, that edge of the window will remain fixed and any addition or reduction in that dimension will be performed on the opposite side of the window.
2. If WINDOW has an EXTENT property and the newly exposed window area is outside it, any extra area will be added so as to show extent that was not previously visible.
3. The current X,Y position is kept visible, if it was visible before the reshape.

5.5.12 INVERTING A WINDOW

A window's contents may be inverted (e.g., change its background texture) using the function **INVERTW**.

Function:	INVERTW
# Arguments:	2
Arguments:	1) WINDOW, a window handle
	2) SHADE, a texture handle
Value:	The window handle.

INVERTW fills the window with the specified texture in inverted mode. If SHADE is NIL, the default value of BLACKSHADE will be used. INVERTW returns the window handle so that it can be used inside a RESETFORM. Consider the following example:

```
<-(INVERTW LOGOW)
{WINDOW}#74,25554
```

Note that the default shade for inverting a window is BLACKSHADE. The inverted logo window is depicted in Figure 5-23.

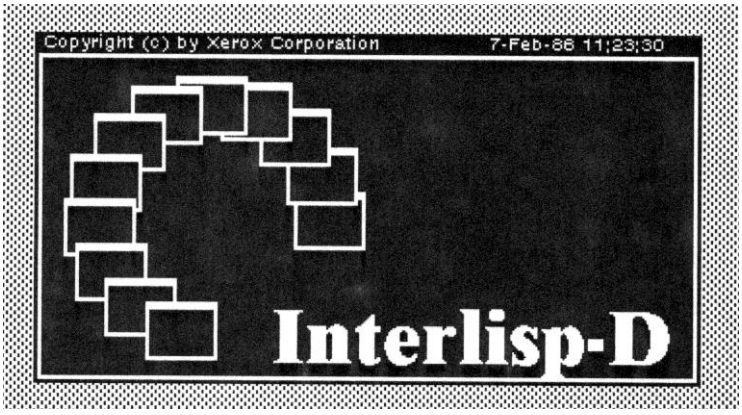

Figure 5-23. Inverted Logo Window with BLACKSHADE

```
<-(INVERTW LOGOW GRAYSHADE)
{WINDOW}#74,25554
```

Inverting a window with GRAYSHADE leaves some residue of structure and form, but these are just barely discernible. This is because the shades chosen for printing the contents of the window correspond to GRAYSHADE. Figure 5-24 depicts an inverted window with GRAYSHADE. You may use GRAYSHADE to indicate a window that is not currently active within your application.

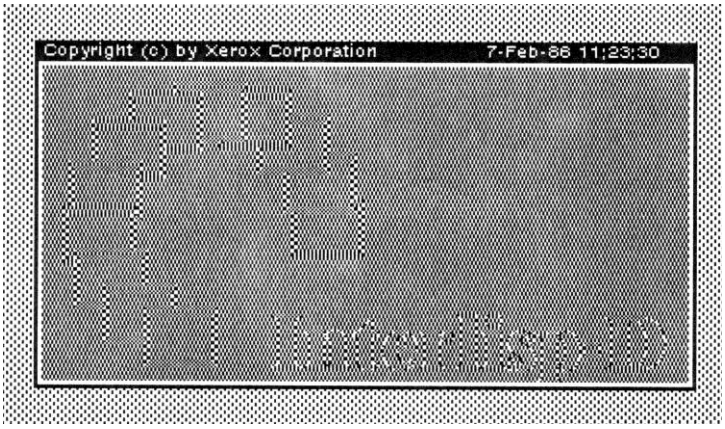

Figure 5-24. Inverting Logo Window with GRAYSHADE

Finally, consider inverting the logo window with a checkerboard pattern which I created using the Bitmap Editor.

```
<-(INVERTW LOGOW CHECKBRUSH)
{WINDOW}#74,25554
```

Note that the contents of the window have been totally obliterated. You will find it useful to experiment with different shades when inverting windows.

One reason for inverting a window is to indicate that its contents are no longer valid (such as when the underlying data structures have been updated) or that the functions associated with that window cannot be performed at this stage of the data processing. Another reason is if one is using multicolored fonts, inverting the window's contents may make it easier to read the contents.

5.5.13 FLASHING A WINDOW

The contents of a window may be *flashed* (i.e., rapidly inverting its contents) as a means of catching the user's attention using the function **FLASHWINDOW**.

Function:	FLASHWINDOW
# Arguments:	4
Arguments:	1) WINDOW, a window handle
	2) N, the number of times to flash
	3) FLASHINTERVAL, the length of time between flashes
	4) SHADE, a texture handle
Value:	NIL.

Generally, you will want to flash a window to alert the user that a particularly important result is being displayed, that some erroneous input has been entered, or that an urgent input is required.

FLASHWINDOW flashes the specified window by inverting it twice. It will flash the window N times. The default number of flashes is 1. FLASHINTERVAL specifies the time to wait in milliseconds between flashes. The default is 200 milliseconds.

When the window is first inverted, it will be given the texture specified by SHADE. If SHADE is NIL, then BLACKSHADE will be used.

If WINDOW is NIL, the entire screen is flashed.

5.5.14 DETERMINING THE MINIMUM WINDOW SIZE

The user may determine the minimum size required by a window using the function **MINIMUMWINDOWSIZE**:

Function:	MINIMUMWINDOWSIZE
# Arguments:	1
Arguments:	1) WINDOW, a window handle
Value:	A dotted pair giving the minimum window size.

MINIMUMWINDOWSIZE returns a dotted pair which is the minimum window size. The CAR of the dotted pair is the minimum width and the CDR is the minimum height. The minimum size of a window is determined by the value of the window property MINSIZE. If the value of MINSIZE is NIL, the default value is a width of 26 and the height required to display the title, border, and one line of text (in the current font). Consider the following example:

```
<-awindow
{WINDOW}#65,23554

<-(MINIMUMWINDOWSIZE awindow)
(26 . 29)

<-(WINDOWPROP awindow 'MINSIZE)
NIL
```

If MINSIZE is a literal atom, it should be the name of a function which is called with WINDOW as its argument. This function should return a dotted pair specifying the minimum size of the window.

5.5.15 OBTAINING A WINDOW FROM A DISPLAY STREAM

Each window has an associated display stream. You may determine the window handle from the display stream using **WFROMDS**:

Function:	WFROMDS
# Arguments:	2
Arguments:	1) DISPLAYSTREAM, a display stream handle 2) DONTCREATE, a flag
Value:	The window handle.

WFROMDS obtains the window handle from the display stream object and returns its as its value. It will return NIL if the destination of the display stream is not a bitmap that supports a window system (for example, it could be a bitmap associated with a printer).

DONTCREATE determines whether or not to create a window if one does not exist in the current set of windows. If T, a window is not created.

WFROMDS is called from TTYDISPLAYSTREAM (see Section 3.6.3.1, II) with DONTCREATE set to T so that it will not create a window unnecessarily. Consider the following examples:

```
<-MYTEXTSTREAM
{STREAM}#64,107554

<-(WFROMDS MYTEXTSTREAM)
{WINDOW}#55,141404
```

5.6 WINDOW PROPERTIES

Windows are represented by objects in the Interlisp system. Each window has a number of properties that record its basic attributes and control its behavior in response to certain events such as the mouse

entering or leaving the window. You may add any arbitrary properties that you want to the existing set of window properties as required by your programs. However, care should be taken not to conflict with existing window property names as many of them are used by the system code.

5.6.1 BASIC WINDOW PROPERTIES

Each window is described by a set of properties established by the system when the window is created. A few of these properties may not be changed by WINDOWPROP.

5.6.1.1 The Window Display Stream

The display stream associated with the window is maintained under the property DSP. All system functions operate on either the window or its display stream. Consider the following example:

```
<-(CREATEW NIL "An Example Window" 3 NIL)
{WINDOW}#56,41404
```

The display stream object address may be obtained via:

```
<-(WINDOWPROP awindow 'DSP)
{STREAM}#65,122000
```

For many display functions, it is possible to specify either the window or its display stream as an argument and achieve the same result.

5.6.1.2 Window Size

The interior dimensions of the window are described by two properties: HEIGHT and WIDTH. The interior space of a window is the space usable for display by the program and does not include the border or title areas. The height and width of the window created above may be obtained via:

```
<-(WINDOWPROP awindow 'HEIGHT)
392

<-(WINDOWPROP awindow 'WIDTH)
392
```

Section 5.5.1.1 described two functions for computing the usable height and width of a window. The function WINDOWSIZE returns a dotted pair consisting of the value of these two properties.

5.6.1.3 The Window Region

The region of a window is the space occupied by the window on the physical screen real estate. The window region includes the title and border areas of the window. It is represented as a region object under the window property REGION. The region of the window created above may be obtained via:

```
<-(WINDOWPROP awindow 'REGION)
(577 0 400 400)
```

5.6.1.4 The Window Title

The window title is a label added to the window to identify the purpose of the window. The title is displayed in the top border of the window when it is open. Otherwise, the title becomes the name of the icon when the window has been collapsed. The window title is maintained under the window property TITLE. The window title for the window created above may be obtained via:

```
<-(WINDOWPROP awindow 'TITLE)
```

"Example Window"

The title of a window is always represented as a string or NIL. If the title is not a string when presented to WINDOWPROP, it coerces it to a string via MKSTRING. The size of the title is determined by the default font (GACHA 10 MRR). The title of a window by providing a third argument to WINDOWPROP. Consider the following example:

```
<-(WINDOWPROP awindow 'TITLE "Steve's Window")
NIL
```

The title of a window by setting the value of TITLE to NIL:

```
<-(WINDOWPROP awindow 'TITLE NIL)
NIL
```

Note that the size of the border at the top of the window shrinks and expands to adjust to the size and length of the title as specified by the window title font.

The background texture of the window title may be set by assigning a value to WINDOWTITLESHADE. It's value is initially BLACKSHADE. Consider the following example:

```
<-(WINDOWPROP W1 'WINDOWTITLESHADE 'CHECKSHADE)
NIL
```

which is depicted in Figure 5.26. Black is always used as the background of the title letters so that they can easily be read. The remaining space is painted with the new title shade.

5.6.1.5 The Window Border

The border of a window is specified as a the width in pixels. Its value is stored under the window property BORDER. You may obtain the border of the window created above via:

```
<-(WINDOWPROP awindow 'BORDER)
4
```

If no border is specified when the window is created, the default border size stored in the system variable,WBorder, is used. Initially, the value of WBorder is 4. You may not specify NIL as the new value of BORDER.

5.6.1.6 The Window Extent

The window extent is the region in the window's display stream that contains the complete image of the object being viewed. The display stream contents may exceed the size of the window. Thus, the user must scroll up and down (perhaps side to side as well) in order to view the complete object. It is kept under the window property EXTENT. The extent of the window created above may be obtained via:

```
<-(SETQ FBW (WHICHW))
{WINDOW}#65,20064

<-(WINDOWPROP FBW 'REGION)
(394 16 316 449)

<-(WINDOWPROP FBW 'EXTENT)
(0 -243 445 684)
```

which indicates that part of the extent would exist off of the screen.

The extent of a window may be NIL whence the window is not scrollable.

Setting the height of the extent to -1 indicates to the Window Manager that you do not know how high the extent will be. The scroll handling functions recognize this situation as meaning that the vertical dimension of the extent is unknown. The same is true for the horizontal parameter for the extent.

5.6.1.7 Assigning the TTY Process

If the PROCESS window property is non-NIL, it will be made the TTY process (see Section 12.1.2,II) upon entry to the window. The value of this property must be a process handle.

Typically, different processes will be assigned to different windows. Whenever a window associated with a process is entered, you want to ensure that that process will respond to keyboard input. This is accomplished by making that process the TTY process for the window.

5.6.1.8 Detecting Page Full Conditions

If the window property PAGEFULLFN is non-NIL, it will be called with the window object as its argument whenever the Window Manager determines that the window is full. A page full condition is detected by the Window Manager when enough characters have been printed in the window such that the next character printed would cause some information to scroll off the top of the window. If PAGEFULLFN is NIL, the system function PAGEFULLFN is invoked.

5.6.2 EVENT PROPERTIES FOR WINDOWS

Event properties of a window are associated with the interactive operations that you may perform on a window by accessing the primary window menu with the right mouse button when the cursor is located in the window.The examples mentioned in the following sections are taken from an inspection of a FileBrowser window.

5.6.2.1 Closing Windows

The event property CLOSEFN records a single function or a list of functions which will be executed when a window is closed by CLOSEW. Each function is called with the window handle as its argument. If any of the functions return the value DON'T, the window will not be closed. To prevent a window from being closed, the atom DON'T may be assigned as the value of CLOSEFN:

```
<-(WINDOWPROP <window> 'DON'T 'CLOSEFN)
NIL
```

When you attempt to close XW either via CLOSEW or from the Standard Window Menu, the Window Manager ignores the request.

None of the functions associated with CLOSEFN should call CLOSEW on its argument as it will enter an infinite loop.

5.6.2.2 Opening Windows

The event property OPENFN records a single function or a list of functions which will be executed after a window is opened by OPENW

The File Browser has as its value for OPENFN a call to two functions:

- It calls OPENATTACHEDWINDOWS to open associated windows for the menu and command panes.
- It calls CLOSEICONWINDOW to close the window displaying the icon that represents this instance of the file browser.

You may prevent a window from being opened by assigning the atom DON'T as the value of OPENFN via:

<-(WINDOWPROP <window> 'DON'T 'OPENFN)
NIL

Note that the value of OPENFN is stored under the USERDATA property associated with the window handle.

5.6.2.3 Bring a Window to the Top

Whenever a window is brought to the top of the occlusion stack (and, therefore, made wholly visible), the function(s) associated with the event property TOTOPFN are executed. These functions may bring other windows to the top of the stack, expand them or open them, or cause them to be redisplayed.

If the argument NOCALLTOPWFN is non-NIL, the function(s) associated with TOTOPFN will not be executed.

The File Browser calls the function TOPATTACHEDWINDOWS to bring to the top the attached windows associated with the display pane of the file browser.

5.6.2.4 Shrinking Windows

The event property SHRINKFN records a single function or a list of functions which will be executed immediately prior to the shrinking of a window into an icon by SHRINKW. If any of these functions returns the value DON'T, the window will not be shrunk.

The File Browser calls the function SHRINKATTACHEDWINDOWS (See Section 5.8,II) as the value of SHRINKFN to shrink the associated attached windows.

You may prevent a window from being shrunk by assigning the value DON'T to SHRINKFN as follows:

<-(WINDOWPROP <window> 'DON'T 'SHRINKFN)
NIL

The PROMPTWINDOW is a window which cannot be shrunk. Inspection of the USERDATA window property will show the atom DON'T as the value of SHRINKFN.

5.6.2.5 Determining a Window's Icon

SHRINKW may be called without specifying the TOWHAT argument. This usually occurs when the *SHRINK* command is invoked from the window menu. In this case, the value of ICONFN is used. This value should be a function which returns a bitmap handle of the icon to be displayed on the screen while the window is shrunk. The function which is the value of ICONFN is called with two arguments:

- The window handle
- A previously created icon bitmap handle, if any

5.6.2.6 Caching an Icon's Bitmap

When an icon is created for a window using a bitmap, the bitmap is cached under the ICON window property of the window with which it is associated.

5.6.2.7 Caching an Icon's Window Handle

When an icon is created for a window, the window handle of the icon is cached under the ICONWINDOW window property of the window with which it is associated. This permits SHRINKW to be called repeatedly while using the same icon. ICONWINDOW is stored in USERDATA.

This icon may be overridden (e.g., redesigned) only by invoking ICONFN explicitly or giving SHRINKW a TOWHAT argument. Consider an example using the Interlisp Logo Window:

```
<-(WINDOWPROP LOGOW 'ICONWINDOW)
{WINDOW}#64,40150
```

The position of the icon is cached under USERDATA under the property ICONPOSITION:

```
<-(WINDOWPROP LOGOW 'ICONPOSITION)
(610 . 715)
```

5.6.2.8 Bringing a Window to the Top

Whenever a window is brought to the top of the obscuration stack, the function assigned to TOTOPFN is invoked. It is given the window handle as its sole argument. One use of this function is to make other windows visible and to bury or shrink windows which are no longer needed.

5.6.2.9 Before and After Moving a Window

Before a window is moved, if its MOVEFN window property is non-NIL, it will be invoked by MOVEW. The value of MOVEFN should be a function or a list of functions that will be called before the Window Manager moves the window. Each function is invoked with two arguments: the window handle and the new position of the lower left corner of the window.

If the value of MOVEFN is the atom DON'T or any of the functions returns the atom DON'T, the window will not be moved.

If the function (respectively, the last one) which is the value of MOVEFN returns a position, the window will be moved to that position rather than the one originally specified.

AFTERMOVEFN, if non-NIL, is a window property which is invoked by MOVEW after a window is moved. Its value is a function or list of functions which are called with the window handle as an argument.

CALCULATEREGION, if non-NIL, is a window property which is used by GETBOXPOSITION. MOVEW calls GETBOXPOSITION if the new region for a window is NIL. GETBOXPOSITION invokes the function which is the value of CALCULATEREGION to determine a region which is used to prompt the user for the position of the window. The function is given the window handle as its argument. If CALCULATEREGION is NIL, the window region is used to prompt the user.

5.6.2.10 Reshaping a Window

When a window is reshaped by SHAPEW, the function or list of functions which are the value of the window property RESHAPEFN are invoked after the window has been reshaped. Each function is called with four arguments:

- the window handle
- a bitmap with the image of the old window in its old shape
- the region within the bitmap that contains the window's old image
- the region of the screen previously occupied by this window

If the value of RESHAPEFN is the atom DON'T, the window will not be reshaped. The default value for RESHAPEFN is the function RESHAPEBYREPAINTFN.

5.6.2.11 Repainting a Window

When a window is redisplayed by REDISPLAYW, the function or functions associated with the window property REPAINTFN will be invoked by the Window Manager with two arguments: the window

handle and the region to be repainted. The region is specified in the coordinates of the window's display stream.

Before the REPAINTFN is called, the clipping region of the window is set to clip all display operations to the area of interest so that the REPAINTFN can redisplay the entire window but only the area of interest will actually be displayed.

The IRM notes that you should not use CLEARW inside a REPAINTFN because it resets the window's coordinate system to the upper left corner. Rather, you should use DSPFILL.

5.6.2.12 The New Region Function

If SHAPEW invokes GETREGION to prompt the user for a region, the value of the window property NEWREGIONFN is passed to GETREGION as its NEWREGIONFN argument.

5.6.2.13 Specifying the Initial Corners

INITCORNERSFN is a window property whose value is a function that returns a list of the initial corners of the ghost region used to prompt the user. This function is given the window handle as its argument.

The ghost region is a list of the form (BASEX BASEY OPPX OPPY) where (BASEX BASEY) specify the anchored corner of the box and (OPPX OPPY) specify the trackable corner.

If SHAPEW calls GETREGION, this function is called to determine the initial ghost region presented to the user to prompt him to specify a region for the window.

5.6.2.14 Shaping a Window

If the window property DOSHAPEFN is non-NIL, its value should be a function which is called by SHAPEW to shape a window. It is called with two arguments: the window handle and the new region.

5.6.2.15 Expanding a Window

When EXPANDW expands a window, it invokes the function or functions which are the value of EXPANDFN after the window is expanded. However, if the value of EXPANDFN is the atom DON'T, the window will not be expanded. Each of the functions which is the value of EXPANDFN is called with the window handle as its single argument.

5.6.3 Mouse Function Window Properties

A number of window properties are associated with responses to mouse activity while the cursor is located in the window. Perhaps the most important are the entry and exit of the mouse to and from the window. Each of these properties takes one or more functions as its value. Each function will be called with the window handle as its argument. These functions are invoked asynchronously, so they perform any terminal input/output from their own window.

The IRM notes that each of these functions should be *self-contained*. Each function should communicate with other functions solely through other window properties. It suggests that these functions should not expect to access variables bound on the stack as the stack context is formally undefined at the time that these functions are called.

5.6.3.1 Entering a Window

Whenever the cursor moves into the window, the CURSORINFN function(s) are executed. Typically, you will use these functions to set variables for the window, perhaps display a menu of operations, and even preposition the cursor to some object in the window.

5.6.3.2 Exiting a Window

Whenever the cursor leaves a window, the CURSOROUTFN function(s) are executed. One possible application is where you have opened a window under program control and moved the cursor to that window. When the cursor leaves the window, you can automatically close it by placing a call to CLOSEW as the value of CURSOROUTFN.

5.6.3.3 Moving the Cursor in a Window

Whenever the cursor moves within a window, the CURSORMOVEDFN function(s) are called.

Tracking the cursor movement allows you to implement the notion of [itactive regions]it within a window. You determine the region by matching the cursor position against some list of regions and performing the appropriate actions associated with each region. Typically, this is to establish various menus based on the region.

5.6.3.4 Assigning the TTY to a Window

Whenever a button is pressed while the cursor is within a window, the WINDOWENTRYFN function(s) are called. The default function for this window property is GIVE.TTY.PROCESS, which gives the process associated with the window the terminal display stream and then invokes the functions which are the value of BUTTONEVENTFN.

5.6.3.5 Handling Mouse Button Events

The BUTTONEVENTFN function(s) are called whenever there is a change in the state of any of the mouse buttons (e.g., moving up or down) while the cursor is located in the window. If another button event occurs while the function(s) associated with BUTTONEVENTFN are running, another event will not occur.

5.6.3.6 Handling the Right Mouse Button

When the right mouse button is pressed while the cursor is in a window, the function(s) associated with RIGHTBUTTONFN will be called in lieu of the function DOWINDOWCOM. This permits you to supply you own right button function while the cursor is in the display pane of the window.

The standard window operation menu for the right button will be invoked whenever the right button is pressed while the cursor is in the title pane of the window.

Note: *If you supply a right button function, you should call DOWINDOWCOM whenever the cursor is not in the interior region of a window.*

5.7 BACKGROUND DISPLAY OPERATIONS

Underlying the windows displayed on the screen is an absolute coordinate system for the display screen. The shade of the ***background*** is a uniform color. Whenever the mouse is in the background, the right mouse button will activate a background menu. The background display menu has been depicted in Figure 5.17.

Because the File Browser and TEdit subsystems have been loaded into this sysout, these subsystems are accessible through the background menu. The following sections describe the standard background display commands.

5.7.1 BACKGROUND OPERATIONS

The background operations that may be performed from the menu depicted in Figure 5.17 are described in Table 5-11.:

Table 5-11. Background Operations

Operation	Description
FileBrowser	Invokes the File Browser (described in Medley Interlisp: Tools and Utilities).
Idle	Causes the system to enter an idle state; options are discussed below.
SaveVM	Causes the current state of the virtual memory to be written to the appropriate disk partition.
Snap	Allows you to save a snapshot of the virtual memory as a file on your external disk. You are prompted for the file name where the snapshot will be written.
Hardcopy	Allows you to obtain a hard copy of a bitmap selected from the display screen. Options are to a file or to a printer.
PSW	Opens a Process Status Window (described in Medley Interlisp: Tools and Utilities).
TEdit	Opens a TEdit window for editing some Lisp object (described in Medley Interlisp: Tools and Utilities).

5.7.1.1 Idle Options

When you select the Idle operation from the background operations menu, you may drag the mouse to the right (through the gray triangle) to obtain a display of the options that are provided by Interlisp, which are described in Table 5-12.

Table 5-12. Idle Operation Options

Option	Description
Show Profile	Displays the current idle profile in the prompt window.
Set Timeout	Prompts the user to set the timeout period. After this interval has expired without any activity by the user (pressing a key, moving the mouse or pressing a mouse button), Interlisp automatically enters idle mode to protect the display screen.
Choose Display	Allows you to choose the form of the idle display.

The structure of the idle profile appears as follows:

```
Allowed Logins:     (<previous user> <anyone>)
Forget:             T
Timeout:            20 minutes
Displayfn:          IDLE.BOUNCING.BOX
SaveVM:             10 minutes
Authenticate:       T
```

Selecting the *Choose Display* option causes Interlisp to prompt you with a menu of the choices for the idle display. Currently, Interlisp supports two choices:

- Bouncing Box
- Bouncing Username

If you choose "Bouncing Username", Interlisp displays the message in the prompt window:

```
New Idle Displayfn:
(LAMBDA (W) (IDLE.BOUNCING.BOX W (USERNAME NIL NIL T)))
```

If the current user name is NIL, then Interlisp uses the symbol "Interlisp" as the default user name. Note that by setting IdleDisplayfn to some other Lambda expression or function, you may create you own idle display.

5.7.2 BACKGROUND VARIABLES

The following variables have corresponding impacts to those discussed in Section 5.6.3:

- BACKGROUNDBUTTONEVENTFN
- BACKGROUNDCURSORINFN
- BACKGROUNDCURSOROUTFN
- BACKGROUNDCURSORMOVEDFN

These variables provide a way of processing cursor action when the cursor is in the background. Each may take the value NIL or a list of one or more functions. When the cursor is in the background and a mouse button changes state, the function(s) associated with the appropriate variable will be executed.

6. MENUS

A ***menu*** is a list of items from which one or more selections may be made. Interlisp uses menus to provide alternative choices in a visible, but easily accessible manner. Rather than typing in the command, you may "mouse" the menu item and cause that selection to be invoked. There are two types of menus that you may define:

1. ***Pop-Up menus***, which appear momentarily while you make your selection, and then disappear. This type of menu is used for infrequent selections so as to avoid screen clutter. An example of this type of menu is the window operations menu that appears when you press the right mouse button while the cursor resides in a window.
2. ***Fixed menus***, which are attached to windows, and are permanently displayed while the window is open. These menus are used to select frequently used operations that affect the window. An example of this type of menu is the Operator Menu associated with the DEDIT window.

6.1 MENU STRUCTURE

A menu consists of two components: a list of items comprising the possible choices and a "when selected" function that is used to process the choice. Menus are represented as datatypes. Menus are created using the **create** command from the record package. The definition of a menu as a DATATYPE is shown below:

```
(DATATYPE MENU
   (IMAGE SAVEIMAGE ITEMS MENUROWS MENUCOLUMNS MENUGRID
    CENTERFLG CHANGEOFFSETFLG MENUFONT TITLE MENUOFFSET
    WHENSELECTEDFN MENUBORDERSIZE MENUOUTLINESIZE WHENHELDFN
    MENUPOSITION WHENUNHELDFN MENUUSERDATA MENUTITLEFONT
    SUBITEMFN MENUFEEDBACKFLG SHADEDITEMS)
   MENUGRID <- (create REGION LEFT <- 0 RIGHT <- 0)
   WHENHELDFN <- (QUOTE DEFAULTMENUHELDFN)
   WHENUNHELDFN <- (QUOTE (CLRPROMPT))
    (ACCESSFNS
      ((ITEMWIDTH
          (fetch (REGION WIDTH)
             of
             (fetch (MENU MENUGRID) of DATUM))
          (replace (REGION WIDTH)
                   of
                   (fetch (MENU MENUGRID) of DATUM)
                   with NEWVALUE))
          (ITEMHEIGHTIDTH
             (fetch (REGION HEIGHT)
                   of
                   (fetch (MENU MENUGRID) of DATUM))
             (replace (REGION HEIGHT)
                   of
                   (fetch (MENU MENUGRID) of DATUM)
                   with NEWVALUE))
          (IMAGEWIDTH
             (fetch (BITMAP BITMAPWIDTH)
                   of (CHECK/MENU/IMAGE DATUM)))
          (IMAGEHEIGHT
             (fetch (BITMAP BITMAPHEIGHT)
                   of (CHECK/MENU/IMAGE DATUM)))
```

```
            (MENUREGIONLEFT
               (IDIFFERENCE
                  (fetch (REGION LEFT)
                     of
                     (fetch (MENU MENUGRID) of DATUM))
                  (fetch MENUOUTLINESIZE of DATUM)))
            (MENUREGIONBOTTOM
               (IDIFFERENCE
                  (fetch (REGION BOTTOM)
                     of
                     (fetch (MENU MENUGRID) of DATUM))
                  (fetch MENUOUTLINESIZE of DATUM)))
         ))
   (SYSTEM))
```

You may manipulate this definition using DEdit on the atom MENU. Consider a menu for selecting a color from a set of colors. We define the menu as follows:

```
<-(SETQ      colormenu
             (create      MENU
                          ITEMS <- '(RED BLUE GREEN YELLOW ORANGE)
                          CENTERFLG <- T
                          MENUCOLUMNS <- 2
                          MENUFONT <- (FONTCREATE 'HELVETICA 10 'BOLD)
                          TITLE <- "Steve's Color Menu
             ))
{MENU}#61,176204
```

will create the following menu when it is displayed on the screen (see Figure 6-1):

Figure 6-1. A Sample Menu

In this case the selection function has not been explicitly defined, so Interlisp will assume the default selection function, DEFAULTWHENSELECTEDFN.

6.1.1 REPRESENTING MENUS

A menu is represented by an Interlisp record. The structure of a menu record can be examined using the Inspector. The structure of the menu created above is presented in Table 6-1.

Table 6-1. Sample Menu Description

Field	Value
ITEMWIDTH	60
ITEMHEIGHT	12

IMAGEWIDTH	122
IMAGEHEIGHT	47
MENUREGIONLEFT	0
MENUREGIONBOTTOM	0
IMAGE	{WINDOW}#61,52470
SAVEIMAGE	NIL
ITEMS	(RED BLUE GREEN YELLOW ORANGE)
MENUROWS	3
MENUCOLUMNS	2
MENUGRID	(1 1 60 12)
CENTERFLAG	T
CHANGEOFFSETFLG	NIL
MENUFONT	{FONTDESCRIPTOR}#70,171260
TITLE	"Steve's Color Menu"
MENUOFFSET	(0 . 0)
WHENSELECTEDFN	NIL
MENUBORDERSIZE	0
CHANGEOFFSETFLG	NIL
MENUFONT	{FONTDESCRIPTOR}#70,171260
TITLE	"Steve's Color Menu"
MENUOFFSET	(0 . 0)
WHENSELECTEDFN	NIL
MENUBORDERSIZE	0
MENUOUTLINESIZE	1
WHENHELDFN	DEFAULTMENUHELDFN
MENUPOSITION	NIL
WHENUNHELDFN	CLRPROMPT
MENUUSERDATA	NIL
MENUTITLEFONT	NIL
SUBITEMFN	NIL
MENUFEEDBACKFLG	NIL
WHENSELECTEDFN	NIL
SHADEDITEMS	NIL

6.2 MENU PROPERTIES

Interlisp provides a large number of properties with which you may customize menus according to the needs of your applications. These properties are set when you specify the menu via the create statement.

6.2.1 ITEM LIST

At a minimum, you must provide a list of items that comprises the menu. The items represent the entries of the menu which are selectable by the user. Menu items may be any size. However, the window in which the menu is displayed is adjusted to accommodate the largest menu item. Because a menu is supposed to prompt you for a selection, menu items should be both terse and mnemonic.

The item list is assigned to the ITEM property when the menu is created. The individual items may any of the following:

1. an atom
2. a list (whose CAR will be displayed)
3. a bitmap (whose image will be displayed)

If an individual item is a list, its format is interpreted as described in Section 6.2.1.1. In the example given, the items are the colors RED, BLUE, GREEN, YELLOW, and ORANGE. Figure 6.2 depicts a menu consisting of bitmaps. This menu is created by the following expression:

```
(create MENU ITEMS <-
    (LIST
      (LIST SELECT-BIT-MAP 'Select "Selects a non-rectangular shape")
      (LIST TEXT-BIT-MAP 'Text "Types text at cursor")
      (LIST STRAIGHT-LINE-BIT-MAP 'Straightline "Draws a straight line")
      (LIST DIAGONAL-LINE-BIT-MAP 'Diagonalline "Draws a diagonal line")
      (LIST BOX-BIT-MAP 'Box "Draws a rectangular shape")
      (LIST ELLIPSE-BIT-MAP 'Ellipse "Draws an ellipse")
      (LIST CIRCLE 'Circle "Draws a circle")
      (LIST CURVE-BIT-MAP 'Curve "Draws a curve")
      (LIST POLYGON 'Polygon "Draws a polygon"))
        MENUCOLUMNS <- 1
    WHENSELECTEDFN <- (FUNCTION DRAW.OPS.FN)
    ))
```

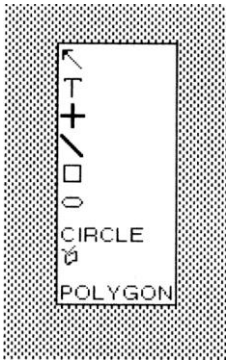

Figure 6-2. A menu comprised of bitmaps

6.2.1.1 Item Specifications

Typically, the item list is just a list of atoms which become the choices of the menu. You may customize the item list in two ways. First, if you want to return a complex value based on the item selected, you may specify an expression which computes the value to be returned. Second, you may specify a prompt string which is displayed in the prompt window when the cursor is placed on the menu item.

The structure of an item specification is:

(<label> <expression> <prompt string>)

where:

label	is displayed on the screen when the menu is accessed.
expression	a form whose value is returned when the item is selected.
promptstring	an explanation string, which is printed in the prompt window, when you press a mouse key with the cursor pointing at this item.

6.2.2 MENU PROCESSING FUNCTION

You may also specify a function that is invoked when an item is selected from the menu. The name or definition (if it is a Lambda expression) is assigned to the property WHENSELECTEDFN. The selection function is called by Interlisp with three arguments:

1. the item selected
2. the menu
3. the mouse key that was pressed to select the item

The general structure of a Lambda function might appear as:

```
(DEFINEQ  (<function> (<item> <menu> <key>)
    (COND
      ((TYPE? <menu> <menu-type for this function>)
        (SELECT <key>
              (LEFT  PROCESS.LEFT.KEY.FOR.<menu>)
               (MIDDLE PROCESS.MIDDLE.KEY.FOR.<menu>)
               (RIGHT PROCESS.RIGHT.KEY.FOR.<menu>)
              (PROGN NIL))
      ))
 ))
```

where each of the PROCESS... functions would have the following structure:

```
(DEFINEQ (PROCESS... NIL
    (SELECTQ <item>
      (<entry-1>    <value-1 or function-1>)
      (<entry-2>    <value-2 or function-2>)
          .....
      (<entry-N>    <value-N or function-N>)
      (PROGN NIL))))
```

You do not have to specify your selection functions in this manner, but I find that it makes the processing code more readable and comprehensible.

We test for the type of menu at the front of the processing function in order to avoid erroneous calls. You may wish to share one processing function across several menus. Then you must test if the function is valid for those menus.

You may interchange the code for key selection versus item selection without affecting the processing of the results. I think it makes the code neater to process the mouse key first.

Note that some of the processing functions may not exist. For example, if it is not possible to use the MIDDLE mouse key with a certain menu, then you would replace PROCESS.MIDDLE.KEY.FOR.<menu> by NIL in the SELECTQ expression above.

6.2.2.1 Default Menu Processing

If you do not supply a menu processing function when you define a menu, Interlisp will use **DEFAULTWHENSELECTEDFN**. This function evaluates <expression>, the second element of an item specification, and returns its value. If there is no second element or the second element is NIL, DEFAULTWHENSELECTEDFN returns the <item> itself.

6.2.3 MENU EXPLANATION FUNCTION

You may specify a menu explanation function that is invoked when a mouse key is held down for an extended duration while the cursor is pointing at a menu item. The function name or its Lambda definition

is stored as the value of the property **WHENHELDFN**. The menu explanation function will be called when the mouse key is held down on an item to which the cursor is pointing for MENUHELDWAIT milliseconds (whose initial value is 1200). The explanation function is called by Interlisp with three arguments:

1. the item selected
2. the menu
3. the mouse key that was pressed to select the item

The general structure of a Lambda function might appear as:

```
(DEFINEQ  (<function> (<item> <menu> <key>)
    (COND
      ((TYPE? <menu> <menu-type for this function>)
        (SELECT        <key>
                      (LEFT  EXPLAIN.LEFT.KEY.FOR.<menu>)
                      (MIDDLE EXPLAIN.MIDDLE.KEY.FOR.<menu>)
                       (RIGHT EXPLAIN.RIGHT.KEY.FOR.<menu>)
                       (PROGN NIL))
      ))
 ))
```

where each of the EXPLAIN.... functions would have the following structure:

```
(DEFINEQ (EXPLAIN... NIL
    (SELECTQ <item>
      (<entry-1>    <value-1 or function-1>)
      (<entry-2>    <value-2 or function-2>)
          .....
      (<entry-N>    <value-N or function-N>)
      (PROGN NIL))))
```

You do not have to specify your explanation functions in this manner, but I find that it makes the processing code more readable and comprehensible.

We test for the type of menu at the front of the processing function in order to avoid erroneous calls. You may wish to share one explanation function across several menus. Then you must test if the function is valid for those menus.

You may interchange the code for key explanation versus item explanation without affecting the processing of the results. I think it makes the code neater to process the mouse key first.

Note that some of the explanation functions may not exist. For example, if it is not possible to use the MIDDLE mouse key with a certain menu, then you would replace EXPLAIN.MIDDLE.KEY.FOR.<menu> by NIL in the SELECTQ expression above.

6.2.3.1 Default Explanation Function

If you do not supply a menu explanation function when you define a menu, Interlisp will use the default explanation function **DEFAULTMENUHELDFN**. This function displays in the prompt window the <promptstring> of an item specification, if it exists. If there is no <promptstring>, the default message "This item will be selected when the button is released." will be displayed instead.

6.2.4 MENU STATUS CHANGE FUNCTION

You may specify a menu status change function when you define a menu that is invoked when you move the cursor from an item in the menu, when another mouse key is pressed, or you release a key which has been held down on an item. The name of this function or its Lambda definition is stored in the property WHENUNHELDFN.

Typically, you will use this function to clear the screen of some display that was presented by the menu explanation function. It is called with the same three arguments as the menu explanation function.

If you do not specify a menu status change function, Interlisp will invoke the default menu status change function, CLRPROMPT. This function merely clears the prompt window.

6.2.5 MENU POSITION

You may specify the menu position on the display screen when the menu is created by MENU or ADDMENU. The menu position is stored in the property MENUPOSITION. There are two cases:

1. If the menu is a pop-up menu, the menu position is the absolute screen coordinates where the menu will be displayed. The value of the property is a position.
2. If the menu is a fixed menu, the value is the coordinates of the window in which the menu appears. The point within the menu image which is placed at these coordinates is specified by the MENUOFFSET property (see Section 6.2.6).

If the value of MENUPOSITION is NIL, then the menu will be displayed at the current cursor position. Usually, you will want to position menus so that they do not obscure the window or object with which they are associated.

6.2.6 MENU DISPLAY OFFSET

You may specify an anchor point for the menu image which will be located at MENUPOSITION. The value of the anchor point is stored in the property MENUOFFSET. The default menu display offset is the position (0,0) in the menu image.

6.2.7 MENU DISPLAY FONT

You may specify a font for the display of items when you define a menu. The value of the font is stored in the property MENUFONT. Chapter 2 describes the types of fonts and their creation. If you do not specify a menu font, Interlisp will use the default font Helvetica 10.

6.2.8 TITLE

You may specify a title when you define a menu. The title is displayed in the top border of the menu window. The value of the title, a string, is stored in the property TITLE. Typically, the menu title will be specified as a string. However, you may also specify a menu title as:

- an atom
- a function or expression which dynamically computes a value

The title is usually displayed in the same font as window titles, when MENUTITLEFONT is NIL. You can change the font in which the menu title is displayed by assigned a new font descriptor to MENUTITLEFONT. I have found that the default menu title font is adequate unless you are going to adjust the size of the title of the menu.

6.2.9 CENTERING MENU ITEMS

You may specify that menu items are centered within the menu window by assigning a non-NIL value to the property CENTERFLG; otherwise, the items are left-justified within the window.

6.2.10 MENU SHAPE

A menu does not have to be a linear list of items. It may also be a matrix of some number of rows and columns. The properties MENUROWS and MENUCOLUMNS specify the number of rows and columns that a menu will have.

Typically, only one of these properties is specified. Interlisp then calculates the other dimension that is necessary to generate a minimum rectangular menu. If you do not specify a shape for a menu by specifying either the number of rows or the number of columns, Interlisp will display the menu items in a single column.

6.2.11 ITEM BOX SIZE

You may specify the size of item boxes when they are displayed in a menu by assigning values to the proerties ITEMHEIGHT and ITEMWIDTH. The item sizes are determined by the font used to display the items. These properties allow you to adjust the size of the menu for easier viewing.

If you do not specify values for either of the properties ITEMHEIGHT or ITEMWIDTH, Interlisp assumes default values. For ITEMHEIGHT, the default value is the maximum height of the following:

1. the height of MENUFONT, or
2. the maximum size of any bitmap appearing as a <label> in the item list

For ITEMWIDTH, the default value is the width of the largest item in the menu.

6.2.12 MENU BORDER SIZE

You may specify a border that will surround an item box. The size of the border is stored in the property MENUBORDERSIZE. If you do not specify a value, 0 (indicating no border) is assumed.

6.2.13 MENU OUTLINE SIZE

A menu is displayed as a window. You may specify a border size for the window surrounding the menu by assigning a value to the property MENUOUTLINESIZE. If you do not specify a value, Interlisp assumes the maximum of:

1. the number 1, or
2. the value of MENUBORDERSIZE

In general, a border enhances the visibility of a menu because it sets it apart from its surrounding area. This is particularly important when the menu "pops up" in the middle of a window filled with text or graphics.

6.2.14 CHANGING MENU OFFSET

Pop-up menus appear wherever the cursor is located on the screen when the menu is created. CHANGEOFFSETFLAG, if non-NIL, determines where the menu appears relative to the cursor by specifying an interpretation for the MENUOFFSET field.

Consider a value for MENUOFFSET of (-1,0). By setting CHANGEOFFSETFLAG to the atom Y, the menu will appear such that the cursor is just to the left of the item last selected. The value of

CHANGEOFFSETFLAG may be the atoms X or Y which specify, respectively, the X and Y coordinates of the MENUOFFSET field.

6.2.15 IMAGE HEIGHT AND WIDTH

Interlisp provides two read-only fields which contain the heighth and width of the menu. These fields are IMAGEHEIGHT and IMAGEWIDTH. They contain the height and width of the menu, respectively.

6.3 MENU MANAGEMENT FUNCTIONS

Interlisp provides a set of functions for managing the creation and display of menus.

6.3.1 CREATING A MENU

You may create a menu using the function **MENU**:

Function:	MENU
# Arguments:	4
Arguments:	1) MENU, the address of a menu object 2) POSITION, a position (in screen coordinates) at which to display the menu 3) RELEASECONTROLFLG, a flag that determines whether the menu is erased 4) NESTEDFLG, a flag that specifies the returned value
Value:	The value computed by the function associated with the WHENSELECTEDFN property.

MENU allows you to ***pop-up*** a menu as needed. The menu appears at POSITION. If POSITION is NIL, the menu appears at the position given by the MENUPOSITION field or the current location of the cursor. Consider the following example:

```
<-(SETQ      colormenu
    (create  MENU
              ITEMS <- '(RED BLUE GREEN YELLOW ORANGE)
             CENTERFLG <- T
             MENUCOLUMNS <- 2
             MENUFONT <- (FONTCREATE 'HELVETICA 10 'BOLD)
             TITLE <- "Steve's Color Menu"))
{MENU}#61,176204
<-(MENU mymenu)
YELLOW
```

When the menu is displayed, MENU waits until you select an item from the menu with a mouse key. Interlisp reverses the video display of the selected item while a key is pressed. When the key is released, the WHENSELECTEDFN function is called with three arguments. WHENSELECTEDFN arguments are:

1. The item selected.
2. The menu.
3. The last mouse key released.

If no item is selected, MENU returns NIL.

In the example above, I placed the cursor on the YELLOW item and pressed the left mouse button. Because the value of WHENSELECTEDFN was DEFAULTWHENSELECTEDFN, the item name was returned as the value of MENU. Once an item is selected, the menu window if closed.

If RELEASECONTROLFLG is T, the menu is not erased after a selection is made.

If NESTEDFLG is T, the Menu handler returns a dotted pair consisting of the item selected and the key which selected it. Consider the following examples:

```
<-(MENU PROPCOPMENU NIL NIL T]
(BT . LEFT)

<-(MENU PROCOPMENU NIL NIL T]
(KBD<- . MIDDLE)
```

6.3.2 ADDING A MENU TO A WINDOW

You may create a permanent menu (i.e., one which exists until you destroy it) using the function **ADDMENU**:

Function:	ADDMENU
# Arguments:	3
Arguments:	1) MENU, a menu object
	2) WINDOW, a window object
	3) POSITION, a position object
Value:	The address of the window object in which the menu is placed.

ADDMENU displays the menu at the specified position in the window. If POSITION is NIL, ADDMENU uses either the value of the MENUPOSITION field in the menu definition or places the menu near the cursor. The menu object's address is assigned to the MENU property of the window. If WINDOW is NIL, a window is created at POSITION (or the defaults) of the size of MENU. This is an example of a ***persistent*** menu.

When you add a menu to a window, the window's CURSORINFN and BUTTONEVENTFN functions are replaced by the value of MENUBUTTONFN. This allows the menu to be active during a TTY waiting period. The RESHAPEFN function is set to restore the menu's image when the window is reshaped.

Item selection proceeds exactly as described for MENU. More than one menu can be assigned to a window. However, a menu can be assigned to only one window at a time. Consider the following example:

```
<-(ADDMENU mymenu)
{WINDOW}#61,146404
```

At anytime, you may close a persistent menu via CLOSEW (see Section 5.5.2) or through the interactive window operations.

6.3.3 DELETING A MENU FROM A WINDOW

DELETEMENU removes a menu from a window. It takes the form:

Function:	DELETEMENU
# Arguments:	3
Arguments:	1) MENU, a menu object
	2) CLOSEFLG, a flag determining whether the window is to be closed or not
	3) FROMWINDOW, a window object
Value:	The window handle.

DELETEMENU removes the menu from the window given by FROMWINDOW. If MENU was the only menu associated with FROMWINDOW, the window will be closed if CLOSEFLG is non-NIL. If FROMWINDOW is NIL, DELETEMENU searches the list of currently active windows for one that contains MENU. If it is found, DELETEMENU acts as above; otherwise, it does nothing. Consider the following example:

```
<-(DELETEMENU (GET.MENU.FROM.WINDOW MW))
{WINDOW}#55,63234
```

which produces the figure below.

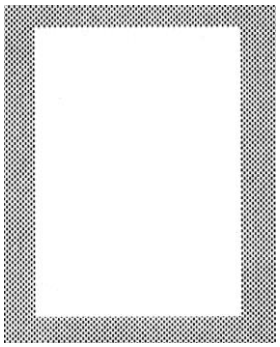

Figure 6-3. Window with Menu Deleted

After you have deleted the menu, you will want to repaint the window with its background shade.

6.3.4 DETERMINING THE WINDOW OF A MENU

You may determine which window a menu is located in using the function **WFROMMENU**:

Function:	WFROMMENU
# Arguments:	1
Arguments:	1) MENU, a menu object
Value:	The address of the window object in which the menu is located; otherwise, NIL.

WFROMMENU returns the address of the window object in which MENU is located. Consider the following example:

```
<-(WFROMMENU mymenu)
{WINDOW}#61,146404
```

You should not attempt to print directly to windows containing menus as you may overwrite portions of the menu.

6.3.5 EXECUTING A MENU ITEM

Sometimes, it is useful to be able to execute the function associated with a menu item without actually displaying the menu and selecting the item. This permits you to define the processing for an item in one function only although you may execute it in several different ways. To execute the function associated with a menu item, you may use the function **DOSELECTEDITEM**:

Function:	DOSELECTEDITEM
# Arguments:	3
Arguments:	1) MENU, a menu object
	2) ITEM, the name of an item in the menu
	3) BUTTON, the name of a mouse key used to select the item
Value:	The value returned by WHENSELECTEDFN given ITEM and BUTTON.

DOSELECTEDITEM invokes the WHENSELECTEDFN of MENU with the values for ITEM and BUTTON. The menu is not displayed on the screen nor is the screen changed in any way. If the specified key is not supported in the selection function, the default value (usually NIL) will be returned by the function associated with WHENSELECTEDFN.

6.3.6 FINDING THE REGION OCCUPIED BY AN ITEM

You may determine the region occupied by an item in a menu using the function **MENUITEMREGION**:

Function:	MENUITEMREGION
# Arguments:	2
Arguments:	1) ITEM, an item of the menu
	2) MENU, a menu object
Value:	A region object.

MENUITEMREGION returns the region occupied by ITEM in MENU when the menu is displayed. Consider the following example:

```
<-(MENUITEMREGION 0 (GET.MENU.FROM.WINDOW MW))
(51 1 50 50)
```

where the menu is depicted in Figure 6.4 below.

6.3.7 SHADING A MENU ITEM

You may shade the region occupied by a menu item using the function **SHADEITEM**:

Function:	SHADEITEM
# Arguments:	4
Arguments:	1) ITEM, an item in the menu
	2) MENU, a menu descriptor
	3) SHADE, a shade handle
	4) DSORW, either a display stream or a window descriptor
Value:	NIL

SHADEITEM shades the region occupied by the item in the menu. This is useful when you want to use a menu for several different purposes but need to indicate that some items are not selectable in certain modes. Consider the following example for shading an item in a number pad:

```
<-(SETQ MW (MAKE.NUMBER.PAD "NUMBER PAD"))

<-(SHADEITEM 7 (GET.MENU.FROM.WINDOW MW) GRAYSHADE MW))
NIL
```

which produces the figure below.

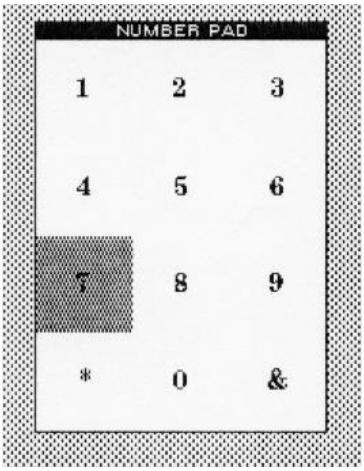

Figure 6-4. Example of Item Shading in a Menu

If DSORW is a display stream or a window, it is assumed to be associated with the window in which the menu is displayed. Otherwise, WFROMMENU is invoked to determine the window in which MENU is located.

6.3.8 Obtaining the Font of a Menu Title

You may obtain the font descriptor of a menu title using the function **MENUTITLEFONT**:

Function:	MENUTITLEFONT
# Arguments:	1
Arguments:	1) MENU, a menu handle
Value:	A font descriptor handle.

This function reads the value of the property MENUTITLEFONT and returns it. Consider the following example:

```
<-(MENUTITLEFONT PROCOPMENU)
{FONTDESCRIPTOR}#70,171540
```

6.3.9 Obtaining the Menu Region

You may obtain the region specification for a menu by executing the function **MENUREGION**:

Function:	MENUREGION
# Arguments:	1
Arguments:	1) MENU, a menu handle
Value:	A region specification.

This function reads the REGION property of the window in which the menu is displayed. Consider the following example:

```
<-(MENUREGION PROCOPMENU)
(0 0 194 50)
```

6.3.10 ERASING A MENU

When you add a menu to a window via ADDMENU, the menu remains permanently displayed. You may erase menus added in this manner using the function **ERASEMENUIMAGE**:

Function:	ERASEMENUIMAGE
# Arguments:	2
Arguments:	1) MENU, a menu handle
	2) WINDOW/STREAM, a window/stream handle
Value:	T

If the menu is displayed in the specified window, its image is erased. Consider the following example:

```
<-(ADDMENU PROCOPMENU AWINDOW)
{WINDOW}#74,25000

<-(ERASEMENUIMAGE PROCOPMENU AWINDOW)
T
```

6.3.11 CREATING A MENU FROM A LIST

In many cases, you will want to create a menu quickly from a list while assuming defaults for all of the menu properties except the items in the menu. You may create menus simply and quickly using the function **TYPEINMENU**, which takes the following form:

Function:	TYPEINMENU
# Arguments:	1
Arguments:	1) LST, a list of items
Value:	A menu handle.

TYPEINMENU creates a menu whose items consist of the items in LST. All other properties assume default values. The menu is displayed at the current location of the cursor. Consider the following example:

```
<-(TYPEINMENU '(A B C D))
{MENU}#61,167260
```

The menu properties take the following values:

```
ITEMWIDTH          11
ITEMHEIGHT         12
IMAGEWIDTH         13
IMAGEHEIGHT        50
MENUREGIONLEFT     0
MENUREGIONBOTTOM   0
IMAGE              {WINDOW}#60,150640
SAVEIMAGE          NIL
ITEMS              (A B C D)
MENUROWS           4
MENUCOLUMNS                1
MENUGRID  (1 1 11 12)
CENTERFLAG                 NIL
```

```
CHANGEOFFSETFLG          NIL
MENUFONT  {FONTDESCRIPTOR}#70,171260
TITLE          NIL
MENUOFFSET       (0 . 0)
WHENSELECTEDFN     UNREADITEM
MENUBORDERSIZE     0
MENUOUTLINESIZE    1
WHENHELDFN       DEFAULTMENUHELDFN
MENUPOSITION       NIL
WHENUNHELDFN       CLRPROMPT
MENUUSERDATA       NIL
MENUTITLEFONT      NIL
SUBITEMFN        NIL
MENUFEEDBACKFLG    NIL
SHADEDITEMS        NIL
```

6.3.12 SELECTING/DESELECTING A MENU ITEM

When you select a menu item using the cursor, the menu item is shaded to indicate that it has been selected. The selection or deselection of a menu item is controlled by the functions **MENUSELECT** and **MENUDESELECT**, respectively. They take the following form:

Function:	MENUSELECT MENUDESELECT
# Arguments:	2 Arguments: 1) ITEM, an item in the menu 2) MENU, a menu handle
Value:	The item selected (MENUSELECT) or NIL (MENUDESELECT).

MENUSELECT shades the item with GRAYSHADE in the menu if it is displayed. You may select as many menu items as you wish for selection in the menu. Consider the following example:

```
<-(ADDMENU PROCOPMENU)
{WINDOW}#74,25000

<-(MENUSELECT 'BT PROCOPMENU)
BT
```

Executing MENUSELECT shades the item in the menu as depicted in Figure 6-5.

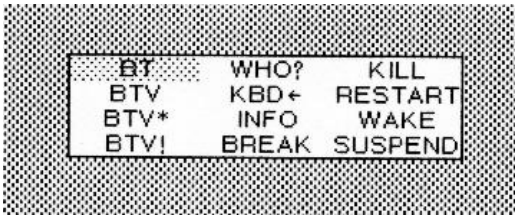

Figure 6-5. Example of MENUSELECT

MENUDESELECT restores the shading of the selected item to WHITESHADE. Consider the following example:

```
<-(MENUDESELECT 'BT PROCOPMENU)
NIL
```

6.3.13 GETTING A MENU ITEM BY GRID COORDINATES

When a menu is displayed on the screen, the Menu Handler internally associates a grid with the items in the menu. When the mouse points to a menu item, the item is determined by the grid coordinates closest to the mouse's position in the menu. You may obtain the menu item corresponding to a grid overlaying a menu using the function **GETMENUITEM**:

Function:	GETMENUITEM
# Arguments:	3
Arguments:	1) MENU, a menu handle
	2) XGRID, the X-coordinate
	3) YGRID, the Y-coordinate
Value:	The menu item closest to the grid coordinates.

Consider the menu PROCOPMENU which is displayed in Figure 6.x with a overlayed on it. The following expressions return items from the menu:

```
<-(GETMENUITEM PROCOPMENU 1 1)
INFO

<-(GETMENUITEM PROCOPMENU 2 0)
SUSPEND
```

Note that the grid begins in the lower left corner with the indices (0,0). Thus, the coordinates (1,1) mean "move right 1 and move up 1". If either of the grid coordinates exceed the grid overlayed on the menu, GETMENUITEM should return NIL. For Example:

```
<-(GETMENUITEM PROCOPMENU 3 0)
NIL
```

because the maximum grid coordinate in the X-direction is 2.

However, there appears to be a bug in the code such that negative grid coordinates wrap around the grid as follows:

```
<-(GETMENUITEM PROCOPMENU -1 0)
WAKE
<-(GETMENUITEM PROCOPMENU -3 0)
BTV
```

6.4 SOME USEFUL MENUS

This section describes some commonly used menus along with the functions for defining and using them.

6.4.1 A YES-NO MENU

A frequently used menu allows you to respond to a question with a YES or NO answer. The question can be displayed in the title pane of the menu. YES and NO becomes the individual items of the menu. The function **YES-NO?** creates the menu for you as shown below:

```
<-(DEFINEQ   (YES-NO? (message)
     (MENU
                (create MENU
                ITEMS <- '((YES T) (NO 'NIL))
                                TITLE <- MESSAGE))))
(YES-NO?)
```

Consider the following example which asks the question “Delete File?”, after you have specified a file.

```
<-(YES-NO? "Delete File:?")
YES
```

where YES was returned as the item selected from the menu.

6.4.2 A NUMBER PAD

A menu which you might find frequent use for is a number pad consisting of the numbers 0 - 9 and the symbols * and #. This menu is depicted in Figure 6-6.

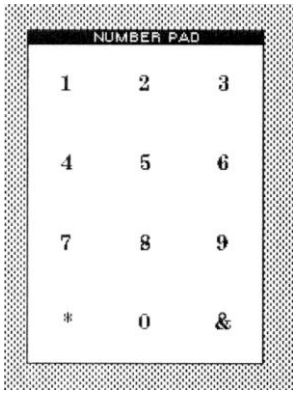

Figure 6-6. A Number Pad Menu

You may define a number pad using the function **MAKE.NUMBER.PAD** as follows:

```
<-(DEFINEQ
   (MAKE.NUMBER.PAD (message)
      (MENU
         (create MENU
               ITEMS <- '(1 2 3 4 5 6 7 8 9 * 0 #)
               TITLE <- MESSAGE
               CENTERFLG <- T
               MENUCOLUMNS <- 3
               ITEMHEIGHT <- 50
               ITEMWIDTH <- 50
               CHANGEOFFSETFLG <- T))))
(MAKE.NUMBER.PAD)
```

When you execute the function MAKE.NUMBER.PAD, it displays the number pad at the location of the cursor. However, the first selection from the menu causes the number pad to disappear. For a persistent number pad, you might modify the function as follows:

```
<-(DEFINEQ
   (MAKE.NUMBER.PAD (message)
      (ADDMENU
         (create MENU
               ITEMS <- '(1 2 3 4 5 6 7 8 9 * 0 #)
               TITLE <- MESSAGE
               CENTERFLG <- T
               MENUCOLUMNS <- 3
               ITEMHEIGHT <- 50
               ITEMWIDTH <- 50
               CHANGEOFFSETFLG <- T))))
(MAKE.NUMBER.PAD)
```

6.4.3 CREATING A FILE OBJECTS MENU

A useful menu is one which allows you to select an object from a file for editing or other operations rather than having to type in its name. I have created the function **FILE.EDIT.MENU** which displays a menu of the types of file package objects. It is defined as:

```
(DEFINEQ
   (FILE.EDIT.MENU (FILE)
   (PROG (OBJECTS OBJECTS-MENU)
      (SETQ OBJECTS '(VARS FNS COMS RECORDS))
      (SETQ OBJECTS-MENU
         (create MENU
               ITEMS <- OBJECTS
               TITLE <- "File Package Objects"
               MENUCOLUMNS <- 1
               CENTERFLG <- T
               WHENSELECTEDFN
                  <- (FUNCTION DISPLAY.OBJECT)))
      (PROMPTPRINT "Select file package object to edit:")
      (MENU OBJECTS-MENU)
      (CLRPROMPT))
   ))
```

I have statically defined the objects to be displayed for purposes of this illustration. You may want to dynamically compute this value by inspecting the file. Now, if I execute this function, I obtain the menu depicted in Figure 6-7.

```
<-(FILE.EDIT.MENU 'SHK)
SQUARE
```

Figure 6-7. File Package Objects Menu

If I select FNS to obtain a list of the file's functions (assuming they are stored a variable of the form <file>FNS), I obtain the menu depicted in Figure 6-8.

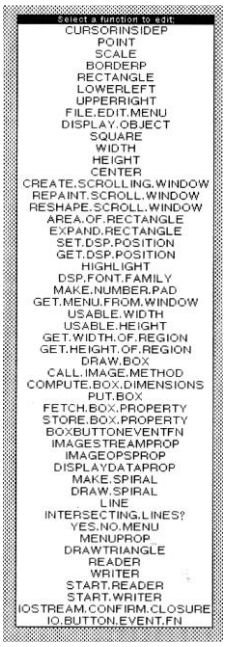

Figure 6-8. Example Menu after Selecting FNS

I may then select one of the functions for editing.

The function **DISPLAY.OBJECT**, which displays all objects of a particular type in a file is defined as:

```
(DEFINEQ (DISPLAY.OBJECT (ITEM MENU KEY)
  (PROG NIL
  (* User must select object using LEFT mouse key only!)
  (if  (OR (EQ KEY 'RIGHT) (EQ KEY 'MIDDLE))
     then (RETURN NIL))
  (SELECTQ ITEM
    (VARS
       (MENU
          (create    MENU
                ITEMS <- (FILECOMLST FILE 'VARS)
                TITLE <- "Select variable"
                MENUCOLUMNS <- 1
                CENTERFLG <- T
                WHENSELECTEDFN  <-
                     (FUNCTION  (LAMBDA (ITEM MENU KEY)
                          (APPLY* 'EDITV ITEM)
```

```
                    )
        )))
(FNS
    (MENU
        (create    MENU
                ITEMS <- (FILEFNSLST FILE)
                TITLE <- "Select a function:"
                MENUCOLUMNS <- 1
                CENTERFLG <- T
                WHENSELECTEDFN
                    <- (FUNCTION
                        (LAMBDA (ITEM MENU KEY)
                        (APPLY* 'EDITF ITEM)
                        ))
        )))
(RECORD
    (MENU
        (create    MENU
                ITEMS <- (FILECOMLST FILE 'RECORDS)
                TITLE <- "Select a record:"
                MENUCOLUMNS <- 1
                CENTERFLG <- T
                WHENSELECTEDFN
                    <- (FUNCTION
                        (LAMBDA (ITEM MENU KEY)
                        (APPLY* 'EDITREC ITEM)
                        ))
        )))
(COMS
    (EDIT (FILECOMS FILE)))
(T
    (PROMPTPRINT "Unknown File Package Object")))
)))
```

7. IMAGE STREAMS

Interlisp has abstracted the notion of a graphics object as a data structure called an ***image object***. An image object contains information about an image, including how to display it, how to print it, and how to manipulate it when it is included in a collection of images.

Image objects were created to provide a mechanism whereby images might be inserted into TEdit files. This permits you to mix text and graphics in a single file where each data type is manipulated by its own editor. When the file is printed, the text and graphics are displayed as a single integrated document.

You communicate with the image object by calling a standard set of functions. These functions are defined by the IMAGEFNS data type, which is a vector of procedures that defines the interfaces from a standard function name to a specific procedure for a specific image object. By grouping the image functions into a data type, multiple instances of the same type of image object can share the same vector of procedures.

Image objects have been defined for several of the other Interlisp data structures in Library packages.

7.1 THE STRUCTURE OF AN IMAGE OBJECT

Image objects are described by the IMAGEOBJ data type which has the following fields (with simple values):

```
<-(IMAGEOBJCREATE 'X)
{IMAGEOBJ}#74,36322
```

Table 7-1. Image Object Data Structure

Field Name	Sample Value
OBJECTDATUM	X
IMAGEOBJPLIST	NIL
IMAGEOBJFNS	{IMAGEFNS}#74,34340

The OBJECTDATUM is the name of the object to be represented. IMAGEOBJPLIST will contain the property list that describes the image object. IMAGEOBJFNS is the vector of functions which permit you to operate upon the object. Initially, the vector will have NIL entries for all functions.

7.1.1 ACCESSING AND SETTING THE IMAGE OBJECT PROPERTIES

You may access and set a subset of the properties of the image object using the function **IMAGEOBJPROP**, which takes the form:

Function:	IMAGEOBJPROP
# Arguments:	3
Arguments:	1) IMAGEOBJECT, an image object handle
	2) PROPERTY, a image object property
	3) NEWVALUE, a new value
Value:	The current value of the property.

IMAGEOBJPROP is a nospread function. IMAGEOBJPROP accesses and sets the properties of the image object. If NEWVALUE is NIL, IMAGEOBJPROP just returns the current value of the property of

the specified object. If NEWVALUE is given, it is assigned as the new value of the property and the current value is returned. IMAGEOBJPROP will work on the following image object properties:

OBJECTDATUM
DISPLAYFN
IMAGEBOXFN
PUTFN
GETFN
COPYFN
BUTTONEVENTFN
COPYBUTTONEVENTFN
WHENOPERATEDONFN
PREPRINTFN

It may also be used to save arbitrary properties that you define for your applications on the image object. Consider the following examples:

```
<-(IMAGEOBJPROP BOX 'DISPLAYFN)
DRAW.BOX
```

7.2 CREATING AN IMAGE OBJECT

You may create an instance of an image object using the function **IMAGEOBJCREATE**:

Function:	IMAGEOBJCREATE
# Arguments:	2
Arguments:	1) OBJECTDATUM, an arbitrary data object
	2) IMAGEFNS, an operations vector
Value:	an image object handle.

IMAGEOBJCREATE creates an instance of an image object and returns its handle. The operations vector for the image functions is associated with the image object instance. Consider the following examples:

```
<-(SETQ boxdemo (IMAGEOBJCREATE 'BOX box))
{IMAGEOBJ}#74,36322
```

which has the following form:

OBJECTDATUM	BOX
IMAGEOBJPLIST	NIL
IMAGEOBJFNS	{IMAGEFNS}#74,34400

where the IMAGEFNS object is created by an expression described in Section 7.4 below.

7.2.1 TESTING FOR AN IMAGE OBJECT

You may test whether an arbitrary Interlisp object is an image object using the function **IMAGEOBJP**:

Function:	IMAGEOBJP
# Arguments:	1
Arguments:	1) X, an arbitrary object
Value:	X, if it is an image object; otherwise, NIL.

IMAGEOBJP returns X if X is an image object; otherwise, it returns NIL. Consider the following example:

```
<-(IMAGEOBJP boxdemo)
{IMAGEOBJ}#74,36322
```

7.3 THE STRUCTURE OF AN IMAGE FUNCTIONS OBJECT

An *image functions object* is a vector of function handles that manipulate an object in an image stream. Typically, the object is displayed on the display screen whence the image functions are used to manipulate its representation interactively. The functions contained within an image functions object are described in Table 7.2. The format of each function and an example are discussed in the following sections.

7.4 CREATING AN IMAGE FUNCTIONS OBJECT

You may create an image functions object using the function **IMAGEFNSCREATE**:

Function:	IMAGEFNSCREATE
# Arguments:	13
Arguments:	1) DISPLAYFN, a display function 2) IMAGEBOXFN, a function that returns the size of the box enclosing the image 3) PUTFN, a function to save the image object on a file 4) GETFN, a function to load an image object from a file 5) COPYFN, a function to implement the copy-select operation 6) BUTTONEVENTFN, a function invoked when a mouse button is pressed while the cursor is in the object 7) COPYBUTTONEVENTFN, a function called when you press a button while the cursor is in the object and the copy key is held down 8) WHENMOVEDFN, a function which is called when the object is moved 9) WHENINSERTEDFN, a function which is called when the object is inserted into a document 10) WHENDELETEDFN, a function which is called when you attempt to delete the object 11) WHENCOPIEDFN, a function which is called when you attempt to copy the object 12) WHENOPERATEDONFN, a function which is called when you attempt to edit the object 13) PREPRINTFN, a function which is called when you attempt to convert the object for printing
Value:	An image functions object handle.

IMAGEFNSCREATE creates an image functions object and returns its handle. Each of the arguments should be the name of a function which is assigned to the corresponding field of the object. The purpose of these function is described in Section 7.5. Consider the following example:

```
<-(SETQ BOX
    (IMAGEFNSCREATE      (FUNCTION DRAW.BOX)
                         (FUNCTION COMPUTE.BOX.DIMENSIONS)
                          (FUNCTION PUT.BOX))
{IMAGEFNS}#74,34400
```

which creates a vector with the following form:

DISPLAYFN	DRAW.BOX
IMAGEBOXFN	COMPUTE.BOX.DIMENSIONS
PUTFN	PUT.BOX
GETFN	NIL
COPYFN	NIL
BUTTONEVENTINFN	NIL
COPYBUTTONEVENTINFN	NIL
WHENMOVEDFN	NIL
WHENINSERTEDFN	NIL
WHENDELETEDFN	NIL
WHENCOPIEDFN	NIL
WHENOPERATEDONFN	NIL
PREPRINTFN	NIL
IMAGECLASSNAME	NIL

This data structure will be used in the examples throughout the remainder of this section as we discuss the different methods associated with an image object. The values for these different functions are set by specifying the values in the calling function. In addition, some of these functions may be set using IMAGEOBJPROP.

7.4.1 TESTING FOR AN IMAGE FUNCTIONS OBJECT

You may test whether an arbitrary Interlisp object is an image functions object using the function **IMAGEFNSP**, which takes the form:

Function:	IMAGEFNSP
# Arguments:	1
Argument:	1) X, an arbitrary Interlisp object
Value:	X, if it is an image functions object; otherwise, NIL.

IMAGEFNSP returns X if X is an IMAGEFNS object; otherwise, it returns NIL. Consider the following example:

```
<-(IMAGEFNSP BOX)
{IMAGEFNS}#74,34614
```

7.5 IMAGE OBJECT FUNCTIONS

The image functions object is a vector of standard methods for operating upon image objects. There are thirteen standard functions that are currently defined by Interlisp. Each of these functions may be passed a ***host stream*** argument, which is the handle of the stream in which the image object is located. Currently, a host stream may be one of:

1. a TEdit *text stream* handle
2. a File Package *file stream* handle
3. an arbitrary Interlisp object handle

The following sections will use the two objects BOXDEMO and BOX which were created in Sections 7.2 and 7.4 above. I have written the following function for calling an image method:

```
(DEFINEQ (call.image.method (imageobj operation window/file/stream)
    (PROG (opfn image.function)
    (RETURN
      (SELECTQ operation
```

```
(DISPLAY
   (PROGN
      (SETQ opfn (IMAGEOBJPROP imageobj 'DISPLAYFN))
      (COND
         ((NULL opfn) NIL)
         (T (EVAL (LIST opfn
                  imageobj
                  (WINDOWPROP WINDOW/FILE/STREAM 'DSP)
                  NIL
                  NIL))
         ))
   ))
(IMAGEBOX
   (PROGN
      (SETQ opfn (IMAGEOBJPROP imageobj 'IMAGEBOXFN))
      (COND
         ((NULL opfn) NIL)
         (T  (EVAL
               (LIST opfn imageobj
                  (WINDOWPROP WINDOW/FILE/STREAM 'DSP)
                  NIL
                  NIL))
         ))
   ))
(PUT
   (PROGN
      (SETQ opfn (IMAGEOBJPROP imageobj 'PUTFN))
      (COND
         ((NULL opfn) NIL)
         (T  (EVAL (LIST imageobj  window/file/stream))
         ))
   ))
```

7.5.1 DISPLAYING AN IMAGE OBJECT

The method for displaying an object is stored in the *DISPLAYFN* field of the image functions object. The function takes the form:

Method:	<DISPLAYFN>
# Arguments:	4
Arguments:	1) IMAGEOBJ, an image object handle 2) IMAGESTREAM, an image stream handle 3) IMAGESTREAMTYPE, the type of the image stream 4) HOSTSTREAM, a host stream handle
Value:	A value returned by the function.

The DISPLAYFN method is invoked to display the image object at the current position of the image stream. The value of IMAGESTREAMTYPE specifies whether it is a display or other type of image stream. When the DISPLAYFN method is called, the offset and clipping regions of the stream are set so that the object's image is positioned at (0, 0). Thus, only the image area may be modified.

The following example involves drawing a box inside a specified window. The image object is BOXDEMO. Prior to calling the display function, you must set the relevant information

concerning the box as properties of the image object. This may be done using the following set of expressions:

```
<-(IMAGEOBJPROP BOXDEMO 'XCOORD 100)
100

<-(IMAGEOBJPROP BOXDEMO 'YCOORD 100)
100

<-(IMAGEOBJPROP BOXDEMO 'WIDTH 200)
200

<-(IMAGEOBJPROP BOXDEMO 'HEIGHT 200)
200
```

The function DRAW.BOX, which is assigned as the value of the DISPLAYFN method is defined as follows:

```
(DEFINEQ
   (DRAW.BOX (imageobj imagestream imagestreamtype hoststream)
   (PROG (xcoord ycoord width height)
         (SETQ xcoord (IMAGEOBJPROP imageobj 'XCOORD))
         (SETQ ycoord (IMAGEOBJPROP imageobj 'YCOORD))
         (SETQ width  (IMAGEOBJPROP imageobj 'WIDTH))
         (SETQ height (IMAGEOBJPROP imageobj 'HEIGHT))
         (for I from xcoord to (IPLUS xcoord width)
                do
                    (BITMAPBIT imagestream I ycoord 1))
         (for I from xcoord to (IPLUS xcoord width)
                do
                    (BITMAPBIT imagestream
                               I
                               (IPLUS ycoord height)
                               1))
         (for I from ycoord to (IPLUS ycoord height)
                do
                    (BITMAPBIT imagestream I xcoord 1))
         (for I from ycoord to (IPLUS ycoord width)
                do
                    (BITMAPBIT imagestream
                               I
                               (IPLUS xcoord width)
                               1))
         (IMAGEOBJPROP imageobj
                       'OBJREGION
                       (CREATEREGION xcoord ycoord width height))
         (RETURN imageobj))
   ))
```

7.5.2 DETERMINING THE SIZE OF AN IMAGE OBJECT

The method for determining the size of an image object is stored in the *IMAGEBOXFN* field of the image functions object. It takes the form:

Method:	<IMAGEBOXFN>
# Arguments:	4

Arguments:	1) IMAGEOBJ, an image object handle
	2) IMAGESTREAM, an image stream handle
	3) CURRENTX, the X-coordinate of the image
	4) RIGHTMARGIN, the X-coordinate of the right margin
Value:	An IMAGEBOX handle.

The IMAGEBOXFN is invoked to determine the size of an image object. It returns the size as an IMAGEBOX, which is a data structure that describes the image that is displayed in terms of width, height, and descender height. The structure of an IMAGEBOX is:

XSIZE	The width of the image object.
YSIZE	The height of the image object.
YDESC	The position of the baseline.
XKERN	The left edge of the image relative to the baseline.

For characters, YDESC becomes the descent and XKERN is the amount of left kerning.

The IMAGEBOXFN looks at the type of stream to determine the output device if the object's size will change as it is displayed on different devices. CURRENTX and RIGHTMARGIN allow an object to know something about its environment so it can determine its own size.

7.5.3 Storing an Image Object Description on a File

The method for storing an image object on a file is stored in the *PUTFN* field of the image functions object. It takes the form:

Method:	<PUTFN>
# Arguments:	2
Arguments:	1) IMAGEOBJ, an image object handle
	2) FILESTREAM, a file handle
Value:	<user defined>

The PUTFN method is invoked to write out a description of the image object to a file. It prints a description on FILESTREAM which can be read by a corresponding invocation of GETFN to recreate the image object in memory.

In our example the function PUT.BOX will print the characteristics of the object BOX on the specified file. Here is the definition of PUT.BOX:

```
<-(DEFINEQ (PUT.BOX (LAMBDA (IMAGEOBJ FILE)
   (PROG (PLIST IMAGEOBJFNS)
      (SETQ PLIST (fetch IMAGEOBJPLIST of IMAGEOBJ))
      (SETQ IMAGEOBJFNS (fetch IMAGEOBJFNS of IMAGEOBJ))
      (PRINTOUT FILE (LIST (QUOTE GET.BOX)))
      (PRINTOUT FILE
            (LIST 'SETQ
                (IMAGEOBJPROP IMAGEOBJ 'NAME)
                (LIST 'IMAGEOBJCREATE
                    (IMAGEOBJPROP IMAGEOBJ 'NAME)
                    (LIST (QUOTE QUOTE)
                        (fetch DISPLAYFN of IMAGEOBJFNS))
                    (LIST (QUOTE QUOTE)
                        (fetch IMAGEBOXFN of IMAGEOBJFNS)))))
      (for X in PLIST bind (Y <- (CADR PLIST)) by (CDDR X)
         do
         (PRINTOUT FILE
```

```
            (LIST 'IMAGEOBJPROP
                (IMAGEOBJPROP IMAGEOBJ 'NAME)
                (LIST (QUOTE QUOTE) X)
                Y)))
   ))
  ))
(PUT.BOX)
```

7.5.4 READING AN IMAGE OBJECT DESCRIPTION FROM A FILE

The method for reading a description of an image object from a file is stored in the *GETFN* field of the image functions object. It takes the form:

Method:	<GETFN>
# Arguments:	1
Arguments:	1) FILESTREAM, a file handle
Value:	An image object descriptor.

The GETFN method is invoked to read the description of an image object from a file in order to create the image object in memory.

7.5.5 COPYING AN IMAGE OBJECT

The method for copying (e.g., duplicating) an image object is stored in the *COPYFN* field of the image functions object. It takes the form:

Method:	<COPYFN>
# Arguments:	3
Arguments:	1) IMAGEOBJ, an image object handle 2) SOURCEHOSTSTREAM, the image stream handle where the image object is located 3) TARGETHOSTSTREAM, the image stream where the copy is built
Value:	An image object descriptor.

The COPYFN method is invoked when the user executes a copy-select operation. If it returns the literal atom DON'T, copying is suppressed.

7.5.6 HANDLING BUTTON EVENTS IN AN IMAGE OBJECT

The method for handling button events inside an image object is stored in the *BUTTONEVENTINFN* field of the image functions object. It takes the form:

Method:	<BUTTONEVENTINFN>
# Arguments:	8
Arguments:	1) IMAGEOBJ, an image object handle 2) WINDOWSTREAM, a window or stream handle 3) SELECTION, a selection handle 4) RELX, the X-coordinate of the cursor location relative to the object's boundaries 5) RELY, the Y-coordinate of the cursor location relative to the object's boundaries 6) WINDOW, a window handle 7) HOSTSTREAM, the source stream handle 8) BUTTON, the mouse button that was pressed

Value: <user defined>

The BUTTONEVENTINFN method is invoked when you press a mouse button while the cursor is located inside some object displayed on the screen. The associated function should decide whether or not to handle the button event. One option is to track the cursor inside the object as the mouse moves. When the cursor leaves the object, the function should relinquish control.

When this function is called, a button has been pressed. It should support the button-down protocol for descending inside composite objects contained within the current object. If the BUTTONEVENTINFN function returns NIL, TEdit treats the button press as a selection at its level 1.

When the BUTTONEVENTINFN function is called, the window's clipping region and offsets are changed so that the lower left corner of the object has the coordinates (0,0). This restricts changes to the object.

7.5.7 HANDLING BUTTON EVENTS DURING COPYING

The method for handling button events inside an image object while copying it is stored in the *COPYBUTTONEVENTINFN* field of the image functions object. It takes the form:

Method:	<COPYBUTTONEVENTINFN>
# Arguments:	2
Arguments:	1) IMAGEOBJ, an image object handle
	2) WINDOWSTREAM, a window or stream handle
Value:	<user defined>

The COPYBUTTONEVENTINFN is invoked when you press a mouse button while the cursor is inside an image object and you are holding down the copy key.

7.5.8 WHEN MOVING AN IMAGE OBJECT

The method for handling the movement of an object by TEdit is stored in the *WHENMOVEDFN* field of the image functions object. It takes the form:

Method:	<WHENMOVEDFN>
# Arguments:	4
Arguments:	1) IMAGEOBJ, an image object handle
	2) TARGETWINDOWSTREAM, the target window, or
	3) SOURCEHOSTSTREAM, a image stream handle
	4) TARGETHOSTSTREAM, an image stream handle
Value:	<user defined>

The WHENMOVEDFN method is invoked when TEdit performs a MOVE operation on the object. This allows you to perform housekeeping chores related to the object and its environment.

7.5.9 WHEN INSERTING AN IMAGE OBJECT

The method for handling the insertion of an image object is stored in the *WHENINSERTEDFN* field of the image functions object. It takes the form:

Method:	<WHENINSERTEDFN>
# Arguments:	4
Arguments:	1) IMAGEOBJ, an image object handle
	2) TARGETWINDOWSTREAM, the target window or stream handle

3) SOURCEHOSTSTREAM, an image stream handle
4) TARGETHOSTSTREAM, an image stream handle

Value: <user defined>

The WHENINSERTEDFN method is invoked when TEdit inserts a copy of the object in a document. This allows you to perform housekeeping chores associated with the object and its new environment.

7.5.10 WHEN DELETING AN IMAGE OBJECT

The method for handling the deletion of an image object is stored in the *WHENDELETEDFN* field of the image functions object. It takes the form:

Method:	<WHENDELETEDFN>
# Arguments:	2
Arguments:	1) IMAGEOBJ, an image object handle
	2) TARGETWINDOWSTREAM, a window or stream handle
Value:	<user defined>

The WHENDELETEDFN method is invoked when TEdit performs a delete operation on the object. This allows you to perform housekeeping chores associated with the object's environment.

7.5.11 NOTIFYING AN IMAGE OBJECT WHEN COPIED

The method for handling the copying of an image object is stored in the *WHENCOPIEDFN* field of the image functions object. It takes the form:

METHOD:	<WHENCOPIEDFN>
# Arguments:	4
Arguments:	1) IMAGEOBJ, an image object handle
	2) TARGETWINDOWSTREAM, the target window or stream handle
	3) SOURCEHOSTSTREAM, an image stream handle
	4) TARGETHOSTSTREAM, an image stream handle
Value:	<user defined>

The WHENCOPIEDFN method is invoked when TEdit performs a copying operation on the object. It is called in addition to, but after, the COPYFN method described above.

7.5.12 NOTIFYING AN IMAGE OBJECT WHEN OPERATED ON

The method for handling editing operations upon an image object is stored in the WHENOPERATEDONFN field of the image functions object. It takes the form:

Method:	<WHENOPERATEDONFN>
# Arguments:	5
Arguments:	1) IMAGEOBJ, an image object handle
	2) WINDOWSTREAM, a window or stream handle
	3) HOWOPERATEDON, the type of operation
	4) SELECTION, a TEdit selection handle
	5) HOSTSTREAM, an image stream handle
Value:	<user defined>

The WHENOPERATEDONFN method is invoked when an image object has been selected for editing. This allows you to perform any pre-editing activities or to invoke the proper editing program (for example, the bitmap editor).

7.5.13 CONVERTING AN IMAGE OBJECT FOR PRINTING

The method for handling the conversion of an image object to something that can be printed in documents is stored in the *PREPRINTFN* field of the image functions object. It takes the form:

Method:	<PREPRINTFN>
# Arguments:	1
Arguments:	1) IMAGEOBJ, an image object handle
Value:	An Interlisp object that can be printed.

The PREPRINTFN method is invoked to convert the object to something that can be printed by PRIN1 or PRIN2. It returns an object that the receiving window can print to obtain a character representation of the object. Consider the following example: If the value of PREPRINTFN is NIL, then the value of the OBJECTDATUM field is used.

7.6 READING AND WRITING IMAGE OBJECTS FROM/TO FILES

Image objects may be read from and written to files stored on external disks or at other nodes in the network. Interlisp uses HREAD and HPRINT to actually read and write the data to/from the file.

7.6.1 READING AN IMAGE OBJECT

You may read an image object from a file using the function **READIMAGEOBJ**:

Function:	READIMAGEOBJ
# Arguments:	3
Arguments:	1) STREAM, a file stream 2) GETFN, an image construction function 3) NOERROR, a flag to ignore errors
Value:	An image object handle.

READIMAGEOBJ reads the description of an image object from the file which is accessed by STREAM. Reading begins at the current file position. It uses the method specified by GETFN to actually construct the image object. GETFN is first validated by Interlisp.

If GETFN cannot be validated or isn't defined, READIMAGEOBJ returns an *encapsulated image object*. This is an image object that captures all the information safely. It will display on the screen as a rectangle with the legend "Unknown IMAGEOBJ Type" and the name of the required GETFN. Selecting an encapsulated image object causes Interlisp to attempt to read the object from the file again. This allows you to load the necessary code required to construct the object description.

You cannot save encapsulated image objects on files because there is not enough information available to the system to copy the object to the file.

7.6.2 WRITING AN IMAGE OBJECT

You may write an image object to a file using the function WRITEIMAGEOBJ:

Function:	WRITEIMAGEOBJ

# Arguments:	2
Arguments:	1) IMAGEOBJ, an image object handle
	2) STREAM, a file stream
Value:	The file stream handle.

WRITEIMAGEOBJ prints a call to READIMAGEOBJ on the file so that the image object will be reconstructed in memory when it is encountered while reading the file. Then, it invokes the PUTFN method to write a description of the image object given by IMAGEOBJ on STREAM. Consider the following example:

7.7 Copying Image Objects Between Windows

To copy an image object from one window to another window:

1. Place the cursor on the object@
2. Press the copy key on the keyboard@
3. Press the LEFT mouse button@

When this sequence of events occurs, the COPYBUTTONEVENTFN of the source window is invoked. If this window supports *copy-selection*, then the implementing function will track the mouse as it moves the cursor across the display screen. When the button is released, the COPYBUTTONEVENTFN should create an image object from the selected information and invoke COPYINSERT to insert it into the current tty window. COPYINSERT invokes the COPYINSERTFN function of the window to insert the image object. This permits the source and destination windows to independently determine how they will handle the copying of image objects.

If the COPYBUTTONEVENTFN is NIL, the BUTTONEVENTFN will be called instead.

If the COPYINSERTFN is NIL, COPYINSERT will convert the image object to a string by calling the PREPRINTFN method of the image object and insert it by calling BKSYSBUF.

7.7.1 Copying and Inserting Image Objects

You may insert a copy of an image object into the window which currently possesses the TTY using the function **COPYINSERT**:

Function:	COPYINSERT
# Arguments:	1
Arguments:	1) IMAGEOBJ, an image object handle
Value:	The image object handle.

COPYINSERT inserts a copy of the image object specified by IMAGEOBJ in the window which currently possesses the TTY process. If the window has a COPYINSERTFN property, its valued is called with the image object handle and the window handle as arguments.

If no COPYINSERTFN exists for the window, BKSYSBUF is invoked with the result of calling the PREPRINTFN method which is given the image object handle as an argument. BKSYSBUF uses PRIN2 to insert the result into the input stream.

7.8 Image Streams

Interlisp performs all image operations through the use of *device-independent graphics* (DIG) data structures and functions. DIG is implemented through the use of *image streams*, which are identified by the occurrence of an IMAGEOPS field in the data structure. The IMAGEOPS field specifies an IMAGEOPS object which is a vector of meaningful graphics operations.

An *image stream* is an output stream which knows how to process graphical display commands for its associated graphics display device. Each image stream has an *image stream type* which specifies the type of graphic display device that it can process graphics commands for. There are three built-in image types: DISPLAY, INTERPRESS, and PRESS.

The display streams discussed in Section 3.5 are an instance of the image stream of type DISPLAY. Not all graphics operations can be properly executed for all image stream types.

7.8.1 THE STRUCTURE OF AN IMAGE STREAM OBJECT

The structure of an image stream corresponds to that of a display stream in that they share many of the same fields. However, the contents are interpreted differently. The structure appears as depicted in Table 7-2.

Table 7-2. Image Stream Object

Field	Value
CHARSET	0
F10	NIL
FW9	0
CBUFMAXSIZE	512
STRNBOUTFN	\BUFFERED.BOUT
STRMBINFN	\BUFFERED.BIN
EXTRASTREAMOP	NIL
IMAGEDATA	{PRESSDATA}#67,37610
IMAGEOPS	{IMAGEOPS}#67,54624
OTHERPROPS	NIL
ENDOFSTREAMOP	\EOSERROR
OUTCHARFN	\OUTCHARFN.PRESS
CBUFDIRTY	NIL
EOLCONVENTION	0
LINELENGTH	94
DIRTYBITS	0
CHARPOSITION	0
MAXBUFFERS	0
FW8	0
CPAGE	0
BUFFS	NIL
BYTESIZE	8
FW7	0
FW6	0
F5	NIL
F4	NIL
F3	NIL
F2	NIL
F1	{COREFILEINFOBLK}#71,30200
EOFFSET	0
EPAGE	0
VALIDATION	NIL
DEVICE	{FDEV}#77,115000
FULLFILENAME	{LPT}.;1
ACCESSBITS	6
USERVISIBLE	T

USERCLOSABLE	T
MULTIBUFFERHINT	NIL
REVALIDATEFLG	NIL
NONDEFAULTDATEFLG	NIL
CBUFPTR	NIL
EXTENDABLE	T
BOUTABLE	T
BINABLE	T
CBUFSIZE	0
COFFSET	0

7.8.1.1 Fetching/Replacing the Fields of an Image Stream

You may fetch or replace the value of a field of an image stream using the function **IMAGESTREAMPROP**:

Function: IMAGESTREAMPROP
Arguments: 1
Arguments: 1) ARGS, a list of arguments
Value: The value of a field of an image stream object.

IMAGESTREAMPROP is an Nlambda, no spread function. It has the following arguments presented in Table 7-3.

Table 7-3. IMAGESTREAMPROP Arguments

Argument	Usage
INSTANCE	An instance of an image stream structure
FIELD	The name of a field in the image stream structure
VALUE	A new value for the field of an image stream structure (optional)

A sample definition for IMAGESTREAMPROP is:

```
<-(DEFINEQ
   (IMAGESTREAMPROP
   (NLAMBDA ARGS
      (RECORDACCESS  (CADR ARGS)
               (EVAL (CAR ARGS))
               '(DATATYPE STREAM
                     ((COFFSET WORD)
                      (CBUFSIZE WORD)
                      (BINABLE FLAG)
                      (BOUTABLE FLAG)
                      (EXTENDABLE FLAG)
                      (NIL BITS 5)
                      (CBUFPTR POINTER)
                      (NONDEFAULTDATEFLG FLAG)
                      (REVALIDATEFLG FLAG)
                      (MULTIBUFFERHINT FLAG)
                      (USERCLOSEABLE FLAG)
                      (USERVISIBLE FLAG)
                      (ACCESSBITS BITS 3)
                      (FULLFILENAME POINTER)
                      (DEVICE POINTER)
```

```
                (VALIDATION POINTER)
                (EPAGE WORD)
                (EOFFSET WORD)
                (F1 POINTER)
                (F2 POINTER)
                (F3 POINTER)
                (F4 POINTER)
                (F5 POINTER)
                (FW6 WORD)
                (FW7 WORD)
                (BYTESIZE BYTE)
                (BUFFS POINTER)
                (CPAGE WORD)
                (FW8 WORD)
                (MAXBUFFERS WORD)
                (CHARPOSITION WORD)
                (DIRTYBITS WORD)
                (LINELENGTH WORD)
                (EOLCONVENTION BITS 2)
                (CBUFDIRTY FLAG)
                (NIL BITS 5)
                (OUTCHARFN POINTER)
                (ENDOFSTREAMOP POINTER)
                (OTHERPROPS POINTER)
                (IMAGEOPS POINTER)
                (IMAGEDATA POINTER)
                (EXTRASTREAMOP POINTER)
                (STRMBINFN POINTER)
                (STRMBOUTFN POINTER)
                (CBUFMAXSIZE WORD)
                (FW9 WORD)
                (F10 POINTER)
                (CHARSET BYTE))
            (if
               (EQ (LENGTH ARGS) 3)
               then
                  (QUOTE FREPLACE)
               else
                  (QUOTE FFETCH))
            (CADDR ARGS))
   ))
(IMAGESTREAMPROP)
```

This function is not defined in the standard Interlisp sysout.

7.8.2 DEFINING A NEW IMAGE STREAM TYPE

You may define new image stream types by adding their definitions to the value of the system variable IMAGESTREAMTYPES. Its initial value is:

```
(4045
   (OPENSTREAM OPEN4045STREAM)
   (FONTCREATE \CREATE4045FONT)
   (FONTSAVAILABLE \SEARCH4045FONTS)
```

```
    (CREATECHARSET \CREATECHARSET.4045))
-----------------------------------------Deprecated-----------------------------------------------------------
(INTERPRESS
    (OPENSTREAM OPENIPSTREAM)
    (FONTCREATE \CREATEINTERPRESSFONT)
    (FONTSAVAILABLE \SEARCHINTERPRESSFONTS)
    (CREATECHARSET \CREATECHARSET.IP))
(PRESS
    (OPENSTREAM OPENPRSTREAM)
    (FONTCREATE \CREATEPRESSFONT)
    (FONTSAVAILABLE \SEARCHPRESSFONTS)
    (CREATECHARSET \CREATECHARSET.PRESS))
(PRESSDISPLAY
    (FONTCREATE \FONTCREATE.HCPYMODE.PRESS)
    (CREATECHARSET \FONTCREATE.HCPYMODE.PRESS))
(INTERPRESSDISPLAY
    (FONTCREATE \FONTCREATE.HCPYMODE.INTERPRESS)
    (CREATECHARSET \FONTCREATE.HCPYMODE.INTERPRESS))
-----------------------------------------Deprecated-----------------------------------------------------------
(DISPLAY
    (OPENSTREAM
        (LAMBDA (FILE)
            (GETSTREAM
                (CREATEW NIL
                        (COND
                            ((EQ FILE '{LPT})
                                "Display image stream")
                            (T   FILE))))))
    (FONTCREATE \CREATEDISPLAYFONT)
    (FONTSAVAILABLE \SEARCHDISPLAYFONTS)))
```

7.8.3 OPENING AN IMAGE STREAM

You may open an image stream of a particular type using the function **OPENIMAGESTREAM**:

Function:	OPENIMAGESTREAM
# Arguments:	3
Arguments:	1) FILE, a file name
	2) IMAGETYPE, the type of image stream
	3) OPTIONS, an options list
Value:	An image stream handle.

OPENIMAGESTREAM opens an image stream of the type specified by IMAGETYPE, which must be one of the image stream types given in the value of the variable IMAGESTREAMTYPES.

```
<-(SETQ IMS (OPENIMAGESTREAM "Sample Image Stream" 'DISPLAY NIL))
{STREAM}#64,120404
```

7.8.3.1 Interpreting FILE

The image stream is opened on the file specified by FILE, which is interpreted as follows:

- If IMAGETYPE is DISPLAY, then the image stream is opened as a window on the display screen. FILE becomes the title of the window. You are prompted to shape the window on the screen.@
- If FILE is the name of a printer, the graphics commands are stored in a temporary file until the image stream is closed by CLOSEF. Then, the commands are automatically sent to the printer.@

- FILE may be the name of a disk file in which case all graphics commands are stored on the file. The file may be sent to the printer after closing the image stream using SEND.FILE.TO.PRINTER or LISTFILES.

7.8.3.2 Inferring the Image Type

If IMAGETYPE is NIL, the image type is inferred from the file name extension and the EXTENSIONS properties of the list PRINTFILETYPES. These extensions may be interpreted as follows:

*** Deprecated ***

- The extensions IP, IPR, and INTERPRESS indicate a file having an Interpress format.
- The extension PRESS indicates a file having a Press format.

*** Deprecated ***

- If there is no extension (e.g., the file name has the form {LPT}<PRINTERNAME>), then IMAGETYPE assumes the type of the associated printer.
- If FILE has no extensions and is not a printer device (e.g., no {LPT} prefix), then IMAGETYPE uses as a default type the type of the first printer listed in DEFAULTPRINTINGHOST.

7.8.3.3 The Options List

OPTIONS is a list of optional directives organized in property list format that modifies the behavior of the image stream. The following properties are currently defined as presented in Table 7-4.

Table 7-4. Optional Directives

Directive	Usage
REGION	The value of this option is the region on the page that text will fill. The position 0,0 is the lower left corner of the page. It is used to initialize DSPLEFTMARGIN, DSPRIGHTMARGIN, DSPBOTTOMMARGIN, and DSPTOPMARGIN. In its absence, the default value is found in DEFAULTPAGEREGION.
FONTS	The value of this option is a list of fonts that may be used in the image stream. Any font can be used in the image stream (subject to limitations of the physical graphic display device), but certain types of image streams operate more efficiently if their fonts are known in advance of printing.
HEADING	The value of this option is the heading to be placed at the top of each page. The heading is placed there automatically by Interlisp. If NIL, no heading is displayed.

7.8.4 TESTING FOR AN IMAGE STREAM

You may test whether an object is an image stream and also its specific type using the function **IMAGESTREAMP**:

Function:	IMAGESTREAMP
# Arguments:	2
Arguments:	1) X, an image stream handle
	2) IMAGETYPE, an image stream type
Value:	X, if it is an image stream; otherwise, NIL.

IMAGESTREAMP is a nospread function. IMAGESTREAMP tests whether or not X is an image stream if IMAGETYPE is NIL. Consider the following example:

```
<-(IMAGESTREAMTYPE IMS)
{STREAM}#64,120404
```

If IMAGETYPE is non-NIL, then IMAGESTREAMP tests whether or not X is an image stream of the specified type. Consider the following example:

```
<-(IMAGESTREAMTYPE IMS 'DISPLAY)
{STREAM}#64,120404

<-(IMAGESTREAMTYPE IMS '4045)
NIL
```

7.8.5 GETTING THE IMAGE STREAM TYPE

You may obtain the image stream type using the function **IMAGESTREAMTYPE**:

Function:	IMAGESTREAMTYPE
# Arguments:	1
Arguments:	1) STREAM, an image stream handle
Value:	The image stream type of STREAM.

IMAGESTREAMTYPE returns the image stream type of the image stream given by STREAM. Consider the following examples:

```
<-(IMAGESTREAMTYPE IMS)
DISPLAY

<-(IMAGESTREAMTYPE (TTYDISPLAYSTREAM))
DISPLAY
```

7.8.6 TESTING THE TYPE OF AN IMAGE STREAM

You may test the type of an image stream using the function **IMAGESTREAMTYPEP**:

Function:	IMAGESTREAMTYPEP
# Arguments:	2
Arguments:	1) STREAM, an image stream handle 2) TYPE, an image stream type
Value:	T, if the image stream is of TYPE.

Consider the following examples:

```
<-(IMAGESTREAMTYPEP IMS 'DISPLAY)
T

<-(IMAGESTREAMTYPE IMS 'PRESS)
NIL
```

Note that if TYPE is NIL, the default type assumed is DISPLAY.

```
<-(IMAGESTREAMTYPEP ((TTYDISPLAYSTREAM))
T
```

7.9 IMAGE STREAM METHODS

An image stream has an IMAGEOPS field which contains a vector of graphics operations to which the image object will respond when sent the appropriate messages. The structure of an IMAGEOPS object for a Press printer is:

```
----------Deprecated----------
IMCHARSET              NILL
IMWRITEPIXEL           NILL
IMSCALEDBITBLT         \SCALEDBITBLT.PRESS
IMFILLPOLYGON          NILL
IMDRAWPOLYGON          NILL
IMRESET              NILL
IMCLIPPINGREGION       \DSPCLIPPINGREGION.PRESS
IMBITMAPSIZE           \BITMAPSIZE.PRESS
IMBACKCOLOR            NILL
IMCHARWIDTHY           NILL
IMCHARWIDTH            \CHARWIDTH.PRESS
IMSTRINGWIDTH          \STRINGWIDTH.PRESS
IMCOLOR              NILL
IMOPERATION            NILL
IMFONTCREATE           PRESS
IMSPACEFACTOR          \DSPSPACEFACTOR.PRESS
IMBOTTOMMARGIN         \DSPBOTTOMMARGIN.PRESS
IMTOPMARGIN            \DSPTOPMARGIN.PRESS
IMTERPRI             NEWLINE.PRESS
IMSCALE              \PRESSINITA0017
IMMOVETO               \PRESSINITA0039
IMNEWPAGE              NEWPAGE.PRESS
IMBITBLT             \BITBLT.PRESS
IMBLTSHADE             \BLTSHADE.PRESS
IMFILLCIRCLE           \PRESSINITA0013
IMDRAWELLIPSE          \DRAWELLIPSE.PRESS
IMDRAWCURVE            \DRAWCURVE.PRESS
IMDRAWLINE             \DRAWLINE.PRESS
IMLINEFEED             \DSPLINEFEED.PRESS
IMRIGHTMARGIN          \DSPRIGHTMARGIN.PRESS
IMLEFTMARGIN           \DSPLEFTMARGIN.PRESS
IMFONT               \DSPFONT.PRESS
IMYPOSITION            \DSPYPOSITION.PRESS
IMXPOSITION            \DSPXPOSITION.PRESS
IMCLOSEFN              \CLOSEF.PRESS
IMAGETYPE              PRESS
----------Deprecated----------
```

Many of the entries in this sample object have the suffix ".PRESS" because the type of object is an object prepared for a PRESS printer. When you create image objects of your own choosing, you will assign functions corresponding to the uses of the image object within your applications.

7.9.1 THE IMAGE TYPE

The *image type field* specifies the type of image object. It is used to record information used by other functions in Interlisp. Its value can be one of those specified in Table 7-5.

Table 7-5. Image Type Values

Type	Usage

DISPLAY	Indicates the image is to be displayed on the monochrome display screen.
(COLOR DISPLAY)	Indicates the image is to be displayed on a color display screen
	The following values are deprecated.
PRESS	Indicates the image is to be displayed on a PRESS printer.
INTERPRESS	>Indicates the image is to be displayed on an Interpress printer.

7.9.2 FONT SPECIFICATION

You may specify the device for which the appropriate font is to be chosen by assigning a value to IMFONTCREATE. Its value is a device name which is passed to FONTCREATE when fonts are created for the stream. In the example above, the name of the device is PRESS indicating the fonts should be created for a PRESS printer. Because some printers do not support certain fonts, it is important to indicate the type of device to be used to display the image object.

7.9.3 FETCHING/REPLACING IMAGEOPS FIELDS

You may fetch or replace the values of fields of an IMAGEOPS structure using the function **IMAGEOPSPROP**:

Function:	IMAGEOPSPROP
# Arguments:	1
Arguments:	1) ARGS, a variable bound to the list of arguments
Value:	The value of the field.

IMAGEOPSPROP is an Nlambda, nospread function. It requires the following arguments:

- INSTANCE: An instance of an IMAGEOPS structure.
- FIELD: The name of a field in an IMAGEOPS structure.
- VALUE: A new value to be assigned to the field (optional).

A sample definition for IMAGEOPSPROP, which I determined by inspecting the FETCHFN of an INSPECTW associated with an image object might be:

```
<-(DEFINEQ (IMAGEOBJPROP (NLAMBDA ARGS)
    (RECORDACCESS      (CADR ARGS)
                       (EVAL (CAR ARGS))
                       '(DATATYPE IMAGEOPS
                           (IMAGETYPE
                            IMCLOSEFN
                            IMXPOSITION
                            IMYPOSITION
                            IMFONT
                            IMLEFTMARGIN
                           IMRIGHTMARGIN
                            IMLINEFEED
                           IMDRAWLINE
                           IMDRAWCURVE
                           IMDRAWCIRCLE
                           IMDRAWELLIPSE
                           IMFILLCIRCLE
                           IMBLTSHADE
                           IMBITBLT
                           IMNEWPAGE
                           IMMOVETO
                           IMSCALE
```

```
                                IMTERPRI
                                IMTOPMARGIN
                                IMBOTTOMMARGIN
                                IMSPACEFACTOR
                                IMFONTCREATE
                                IMOPERATION
                                IMCOLOR
                                IMSTRINGWIDTH
                                IMCHARWIDTH
                                IMCHARWIDTHY
                                IMBACKCOLOR
                                IMBITMAPSIZE
                                IMCLIPPINGREGION
                                IMRESET
                                IMDRAWPOLYGON
                                IMFILLPOLYGON
                                IMSCALEDBITBLT
                                IMWRITEPIXEL
                                IMCHARSET))
    (if (EQ (LENGTH ARGS) 3)
    then
            (QUOTE FREPLACE)
    else
            (QUOTE FFETCH))
    (CADDR ARGS))
    ))
(IMAGEOPSPROP)
```

This function both fetches and replaces fields of an IMAGEOPS structure. Note that it determines that it must replace the value of a field by examining the length of the argument list. This function is not defined in the standard Interlisp sysout.

7.9.4 IMAGE STREAM METHODS

The methods associated with an image stream are described in the following sections. The value of each method should be a function which is called under the appropriate circumstances as described.

The code in the following sections is meant to suggest how to write the prescribed function. You will have to customize the functions for your individual applications and devices. Note that the value returned by the image stream method depends on the value returned by the function that you have specified as the value of the method.

7.9.4.1 Closing a Stream

The **IMCLOSEFN** method is called before the stream is closed by CLOSEF. It takes the form:

Method:	<IMCLOSEFN>
# Arguments:	1
Arguments:	1) STREAM, a stream handle
Value:	<file name>

This method is used to cleanup the image prior to printing. It may flush buffers, write header or trailer information, or perform other housekeeping tasks. Initially, IMCLOSEFN has the value NILL.

Consider the following definition for the function IM.CLOSE.STREAM. It saves the page buffer for the page that you have been constructing and closes the stream.

```
(DEFINEQ   (IM.CLOSE.STREAM (STREAM)
   (PROG (DD)
            (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
            (SAVE.PAGE.BUFFER (DISPLAYDATAPROP DD 'DDDestination) STREAM)
      (*
         The file information block is stored in F1.
      )
      (CLOSEF (IMAGESTREAMPROP STREAM 'F1))
   )))
```

The function SAVE.PAGE.BUFFER should copy the bits in the page buffer, which is a bit map, to the appropriate file stream. How you do this depends on the device for which you are writing the image stream methods.

7.9.4.2 Drawing a Line

The **IMDRAWLINE** method is used to draw a line in the stream from the point (X1,Y1) to the point (X2,Y2). It takes the form:

Method:	<IMDRAWLINE>
# Arguments:	9
Arguments:	1) STREAM, a stream handle
	2) X1, an integer
	3) Y1, an integer
	4) X2, an integer
	5) Y2, an integer
	6) WIDTH, the width of the line
	7) OPERATION, the bitblt operation
	8) COLOR, the color of the line
	9) DASHING, a dashing flag
Value:	The position (X2,Y2).

IMDRAWLINE typically defaults to calling DRAWLINE. Consider the function IM.DRAW.LINE which draws a line on an arbitrary stream.

```
(DEFINEQ (IM.DRAW.LINE (STREAM X1 Y1 X2 Y2 WIDTH OPERATION COLOR)
   (PROG (DD)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (\CLIPANDDRAWLINE
            (\DSPTRANSFORMX X1 DD)
            (\DSPTRANSFORMY Y1 DD)
            (\DSPTRANSFORMX X2 DD)
            (\DSPTRANSFORMY Y2 DD))
            (if (NULL WIDTH)
               then 1
               else WIDTH)
            (SELECTQ OPERATION
               (NIL (DISPLAYDATAPROP DD 'DDOperation))
               ((REPLACE PAINT INVERT ERASE)
                     OPERATION)
               (\ILLEGAL.ARG OPERATION))
            (DISPLAYDATAPROP DD 'DDDestination)
```

```
        (DISPLAYDATAPROP DD 'DDClippingLeft)
        (SUB1 (DISPLAYDATAPROP DD 'DDClippingRight))
        (DISPLAYDATAPROP DD 'DDClippingBottom)
        (SUB1 (DISPLAYDATAPROP DD 'DDClippingTop))
        STREAM
        COLOR))
  (MOVETO X2 Y2 STREAM)
  ))
```

This function calls the system routine \CLIPANDDRAWLINE to draw lines with the arguments appropriate to the stream. It leaves the cursor at the point (X2,Y2) as if it had finished drawing the line.

7.9.4.3 Drawing a Curve

The **IMDRAWCURVE** method is used to draw a curve in an image stream. It takes the form:

Method:	<IMDRAWCURVE>
# Arguments:	5
Arguments:	1) STREAM, a stream handle
	2) KNOTS, a list of points along the curve
	3) CLOSED, a flag indicating a closed curve
	4) BRUSH, the brush type
	5) DASHING, a flag indicating dashing
Value:	The value of STREAM.

IMDRAWCURVE typically defaults to DRAWCURVE. Consider the function IM.DRAW.CURVE which is defined below.

```
(DEFINEQ   (IM.DRAW.CURVE (STREAM KNOTS CLOSED BRUSH)
    (SELECTQ (LENGTH KNOTS)
        (0  NIL)
        (1  (if     (type? POSITION (CAR KNOTS))
            then
                (DRAWPOINT  (fetch XCOORD OF (CAR KNOTS))
                            (fetch YCOORD OF (CAR KNOTS)))
            else
                (ERROR "Bad knot point:" (CAR KNOTS))
            ))
        (2  (if     (OR   (type? POSITION (CAR KNOTS))
                          (type? POSITION (CADR KNOTS)))
            then
                (\LINEWITHBRUSH
                    (fetch XCOORD of (CAR KNOTS))
                    (fetch YCOORD of (CAR KNOTS))
                    (fetch XCOORD of (CADR KNOTS))
                     (fetch YCOORD of (CADR KNOTS))
                     BRUSH
                    DASHING
                    STREAM
                     \BRUSHBBT)
            else
                (ERROR    "Bad knot points: (LIST  (CAR KNOTS) (CADR KNOTS)))
            ))
           (\CURVE2
            (PARAMETRICSPLINE         KNOTS
```

```
                                        CLOSED
                                        BRUSH
                                        DASHING
                                        \BRUSHBBT
                                        STREAM)
        ))
  STREAM
  ))
```

This function tests for the length of KNOTS. It treats a null list as an empty curve. A single point causes the specified brush shape to be displayed at the point. Two points yields a line connecting them. Otherwise, \CURVE2, an internal system function is called to draw a spline through the points.

7.9.4.4 Drawing a Circle

The **IMDRAWCIRCLE** method is used to draw a circle on an image stream. It takes the form:

Method:	<IMDRAWCIRCLE>
# Arguments:	6
Arguments:	1) STREAM, an image stream handle
	2) CENTERX, the X-coordinate of the circle's center
	3) CENTERY, the Y-coordinate of the circle's center
	4) RADIUS, the radius of the circle
	5) BRUSH, the brush type used to draw the circle
	6) DASHING, a flag controlling dashing
Value:	NIL.

IMDRAWCIRCLE draws a circle of the given radius around the point (CENTERX, CENTERY) in image stream coordinates. It uses DRAWCIRCLE (see Section 12.4.1, II) to actually place the bits. Consider the function IM.DRAW.CIRCLE which has the following possible definition:

```
(DEFINEQ    (IM.DRAW.CIRCLE (STREAM CENTER-X CENTER-Y RADIUS BRUSH)
    (PROG (DD CX CY D X Y)
    (* Test for a proper radius.)
    (if         (OR (NOT (NUMBERP RADIUS)) (ILESSP RADIUS 0))
     then       (\ILLEGAL.ARG.RADIUS)
     elseif     (EQ RADIUS 0)

    then        (RETURN NIL))

     (SETQ DD (fetch IMAGEDATA of STREAM))
    (*  Set up for the system curve drawing function.)
    (SETUP.FOR.\BBTCURVEPT.)
    (SETQ    CX
             (\DSPTRANSFORMX  (IDIFFERENCE CENTER-X (BRUSHWIDTH BRUSH)) DD))
    (SETQ   CY
             (\DSPTRANSFORMY (IDIFFERENCE CENTER-Y (BRUSHHEIGHT BRUSH)) DD))
     (*  Now, if the radius is 1, just put a single brush down.)
    (if         (EQ RADIUS 1)
     then       (PROGN
                      (\CURVEPT CX CY)
                      (RETURN NIL)
    ))
```

```
    (*  Otherwise, draw the top and bottom points of the circle.)
    (\CURVEPT CX (IPLUS CY RADIUS))
    (\CURVEPT CX (IDIFFERENCE CY RADIUS))

LOOP

    (*  Update the points to be drawn.)
    (if         (IGREATERP 0 D)
     then
       (PROGN
              (SETQ X (ADD1 X))
              (if      (IGREATERP (ITIMES (IPLUS D Y) 2) 1)
               then
                      (PROGN
                              (SETQ D (IPLUS D (ITIMES (IDIFFERENCE X Y)2)4))
                             (SETQ Y (SUB1 Y)))
               else
                      (SETQ D (IPLUS D (ITIMES X 2) 1)))
              )
     elseif     (OR (EQ 0 D) (IGREATERP X D))
          then
              (PROGN
                      (SETQ X (ADD1 X))
                       (SETQ D (IPLUS D (ITIMES (IDIFFERENCE X Y) 2) 4))
                      (SETQ Y (SUB1 Y)))
          else
               (PROGN
                      (SETQ D (IPLUS D (ITIMES (IDIFFERENCE X Y)2) 4))
                       (SETQ Y (SUB1 Y))
    ))
    (*   Draw the rest of the curve points.)
    (if   (EQ Y 0)
     then
          (PROGN
              (\CURVEPT (IPLUS CX X) CY)
              (\CURVEPT (IDIFFERENCE CX X) CY))
     else
          (PROGN
              (\CIRCLEPTS CX CY X Y)
              (GO LOOP)
       ))

   (MOVETO CENTER-X CENTER-Y STREAM)
   (RETURN NIL))
   ))
```

7.9.5 DRAWING AN ELLIPSE

The IMDRAWELLIPSE method is used to draw an ellipse on an image stream. It takes the form:

Method:	<IMDRAWELLIPSE>
# Arguments:	8
Arguments:	1) STREAM, an image stream handle
	2) CENTERX, the X-coordinate of the center point of the ellipse
	3) CENTERY, the Y-coordinate of the center point of the ellipse

4) SEMIMINIRRADIUS, the minor axis radius
5) SEMIMAJORRADIUS, the major axis radius
6) ORIENTATION, the angular orientation of the ellipse
7) BRUSH, the type of brush to be used
8) DASHING, a flag controlling dashing

Value: NIL.

IMDRAWELLIPSE draws an ellipse of the given proportions (e.g., with the radii given by SEMIMINORRADIUS and SEMIMAJORRADIUS) about the point (CENTERX, CENTERY) in image stream coordinates with the specified orientation. It uses DRAWELLIPSE to actually lay down the bits. Consider the following definition for IM.DRAW.ELLIPSE:

```
(DEFINEQ
  (IM.DRAWELLIPSE (STREAM CENTER-X CENTER-Y SEMIMINOR
            SEMIMAJOR ORIENTATION BRUSH)
    (PROG (CX CY DD X1 X2 Y1 Y2 COS.OF.ORINETATION
          SIN.OF.ORIENTATION MINOR2 MAJOR2 COS2
          SIN2 CY+ CY- Y-OFFSET C1 C2 A1 B1 D)
    (*  C1, C2 ARE CURVATURE TERMS )
    (if  (OR (EQ 0 SEMIMINOR) (EQ 0 SEMIMAJOR))
        then
        (PROGN
            (MOVETO CENTER-X CENTER-Y STREAM)
            (RETURN NIL)
        ))
    (*  Validate the semiminor and semimajor radii.)
    (if (ILESSP SEMIMINOR 1)
        then
            (\ILLEGAL..ARG.SEMINMINORRADIUS)
        elseif
            (ILESSP SEMIMAJOR 1)
            then
                (\ILLEGAL.ARG.SEMIMAJORRADIUS)
        elseif
            (OR (NULL ORIENTATION) (EQ SEMIMINOR SEMIMAJOR))
            then
                (SETQ ORIENTATION 0)
        elseif
            (NULL (NUMBERP ORIENTATION))
            then
                (\ILLEGAL.ARG.ORIENTATION))

    (SETQ COS.OF.ORIENTATION (COS ORIENTATION))
    (SETQ SIN.OF.ORIENTATION (SIN ORIENTATION))
    (SETQ COS2 (FTIMES COS.OF.ORIENTATION COS.OF.ORIENTATION))
    (SETQ SIN2 (FTIMES SIN.OF.ORIENTATION SIN.OF.ORIENTATION))
    (SETQ MINOR2 (ITIMES SEMIMINOR SEMIMINOR))
    (SETQ MAJOR2 (ITIMES SEMIMAJOR SEMIMAJOR))

    (.SETUP.FOR.\BBTCURVEPT.)
    (*  Establish the center points.)
    (SETQ DD (fetch DISPLAYDATA of STREAM))
    (SETQ CX
        (\DSPTRANSFORMX
            (IDIFFERENCE CENTER-X (BRUSHWIDTH BRUSH))
            DD))
```

```
    (SETQ CY
        (\DSPTRANSFORMY
           (IDIFFERENCE CENTER-Y (BRUSHHEIGHT BRUSH))
           DD))
    (SETQ C1 (FPLUS 2 (FTIMES MAJOR2 COS2) (FTIMES MINOR2 SIN2)))
    (SETQ C2
        (FTIMES
           (FPLUS (FTIMES MINOR2 COS2) (FTIMES MAJOR2 SIN2))
           8))
    (SETQ G
        (FTIMES  COS.OF.ORIENTATION
              SIN.OF.ORIENTATION
              (TIMES (DIFFERENCE MINOR2 MAJOR2) 2)
        ))
    (SETQ Y-OFFSET (QUOTIENT (TIMES SEMIMINOR SEMIMAJOR) (SQRT A)))
    (SETQ CY+ (IPLUS CY Y-OFFSET))
    (SETQ CY- (IDIFFERENCE CY Y-OFFSET))
    (SETQ U (TIMES (FTIMES C1 (TIMES Y-OFFSET 2)) 2))
    (SETQ V (TIMES (FTIMES G Y-OFFSET) 4)))
    (SETQ K
        (TIMES
            (DIFFERENCE
                 (TIMES MINOR2 MAJOR2)
                 (FTIMES C1 (TIMES Y-OFFSET Y-OFFSET)))
            4))
    (SETQ C1 (TIMES C1 8))
    (SETQ G (TIMES G 4))
    (*    Begin drawing in Octant 1)
    (SETQ X1 1)
    (SETQ Y1 0)

    (if  (MINUSP V)
        then
        (PROGN
            (SETQ Y2 1)
            (SETQ B1
                (IMINUS
                   (IPLUS V
                        (IQUOTIENT C1 8)
                        (HALVE U))))
            (SETQ A1 (IDIFFERENCE U B1))
            (SETQ D
                (IPLUS B1
                     (QUOTIENT C1 8)
                     (IMINUS (IPLUS K (HALVE U)))
                ))
        ))

MOVE
    (if
        (MINUSP D)
      then
        (PROGN
            (SETQ X (IPLUS X X1))
            (SETQ Y (IPLUS Y Y1))
            (SETQ D (IPLUS B1 D)))
```

```
    else
      (PROGN
          (SETQ X (IPLUS X X2))
          (SETQ Y (IPLUS Y Y2))
          (SETQ D (IDIFFERENCE D A1))
  ))

  (if
      (MINUSP X)
   then
      (PROGN
          (MOVETO CENTER-X CENTER-Y STREAM)
          (RETURN NIL
   ))

  (\CURVEPT (IPLUS CX X) (IPLUS CY+ Y))
  (\CURVEPT (IDIFFERENCE CX X) (IDIFFERENCE CY- Y))
  (if
      (MINUSP B1)
      then
          (GO SQUARE-MOVE))

DIAGONAL-MOVE
  (*
      Make a diagonal transition in an octant
  )
  (OR (MINUSP A1) (GO MOVE))
  (SETQ X1 (IDIFFERENCE X2 X1))
  (SETQ Y1 (IDIFFERENCE Y2 Y1))
  (SETQ B1 (IPLUS B1 A1))
  (SETQ D (IPLUS B1 (IMINUS D) (IMINUS (HALVE (ADD1 A1)))))
  (OR (MINUSP B1) (GO MOVE))

SQUARE-MOVE
  (*
      Make a vertical/horizontal transition in an octant
  )
  (if
      (ZEROP X1)
   then
          (SETQ X2 (IMINUS X2))
   else
          (SETQ Y2 (IMINUS Y2)))

  (SETQ D (IDIFFERENCE (IDIFFERENCE B1 A1) D))
  (GO DIAGONAL-MOVE))
```

7.9.6 FILLING A POLYGON

The **IMFILLPOLYGON** method is used to fill an arbitrary polygon with a given texture. It takes the form:

```
    Method:             <IMFILLPOLYGON>
    # Arguments:        3
```

Arguments:	1) STREAM, an image stream descriptor 2) POINTS, a list of points describing the polygon 3) TEXTURE, a texture object handle
Value:	<user-defined>

IMFILLPOLYGON shades the arbitrary polygon described by POINTS with the specified texture.

7.9.7 FILLING A CIRCLE

The IMFILLCIRCLE method is used to fill a circle with a given texture. It takes the form:

Method:	<IMFILLCIRCLE>
# Arguments:	5
Arguments:	1) STREAM, an image stream descriptor 2) CENTERX, the X-coordinate of the center of the circle 3) CENTERY, the Y-coordinate of the center of the circle 4) RADIUS, the radius of the circle 5) TEXTURE, a texture object
Value:	<user-defined>

IMFILLCIRCLE fills a circle of a given radius whose center is located at the point (CENTERX, CENTERY) in the image stream with the specified texture. The function IMFILLCIRCLE might be used to fill a circle:

```
(DEFINEQ
   (IMFILLCIRCLE (STREAM CENTER-X CENTER-Y RADIUS)
   (PROG (OPERATION DD X Y DBM D CX CY TEXTURE)

   (*  DBM is the destination bit map.)

   (if  (OR (NOT (NUMBERP RADIUS)) (ILESSP RADIUS 0))
      then (\ILLEGAL.ARG.RADIUS))

   (SETQ X 0)
   (SETQ Y RADIUS)
   (SETQ D (ITIMES 2 (SUB1 RADIUS)))
   (SETQ DD (fetch IMAGEDATA of STREAM))
   (SETQ OPERATION (fetch DDOPERATION of DD))
   (SETQ DBM (fetch DDDestination of DD))
   (SETQ TEXTURE (fetch DDTexture of DD))

   (SETQ CX (\DSPTRANSFORMX CENTER-X DD))
   (SETQ CY (\DSPTRANSFORMY CENTER-Y DD))

LOOP

   (if  (IGREATERP 0 D)
      then
      (PROGN
         (SETQ X (ADD1 X))
         (if  (IGREATERP (ITIMES (IPLUS D Y) 2) 1)
            then
               (SETQ D (IPLUS (ITIMES (IDIFFERENCE X Y) 2) 4)))
            else
```

```
            (SETQ D (IPLUS D (ITIMES X 2) 1))))
    elseif (OR (ZEROP D) (IGREATERP X D))
        then
        (PROGN
            (SETQ X (ADD1 X))
            (SETQ D (IPLUS (ITIMES (IDIFFERENCE X Y) 2) 4)))
    else
        (SETQ D (IPLUS (IDIFFERENCE D (ITIMES Y 2)) 3)))
(*  Draw the middle line.)
(if  (ZEROP Y)
    then
    (PROGN
        (\FILLCIRCLEBLT CX CY X Y)
        (SETQ Y (SUB1 Y))
        (GO LOOP)
    ))

(MOVETO CENTER-X CENTER-Y STREAM)
(RETURN NIL)
))
```

7.9.8 SHADING THE OBJECT

The **IMBLTSHADE** method is used to shade a region of the specified image stream with the given texture. It takes the form:

Function:	<IMBLTSHADE>
# Arguments:	8
Arguments:	1) TEXTURE, a texture object
	2) STREAM, an image stream descriptor
	3) DESTINATIONLEFT, ??
	4) DESTINATIONBOTTOM, ??
	5) WIDTH, the width of the region to be shaded
	6) HEIGHT, the height of the region to be shaded
	7) OPERATION, a BITBLT operation
	8) CLIPPINGREGION, a region descriptor
Value:	<user-defined>

IMBLTSHADE fills the region within STREAM of the given dimensions (DESTINATIONLEFT, DESTINATIONBOTTOM, WIDTH, HEIGHT) with the given texture. Its effect is mediated both by the clipping region and the operation.

7.9.9 BIT-BLITTING TO THE OBJECT

The **IMBITBLT** method determines how to bit-blit data into the specified stream. It takes the form:

Method:	<IMBITBLT>
# Arguments:	15
Arguments:	1) SOURCEBITMAP, the source bit map handle
	2) SOURCELEFT, the X-coordinate of the bits to be copied
	3) SOURCEBOTTOM, the Y-coordinate of the bits to be copied
	4) STREAM, the destination stream handle
	5) DESTINATIONLEFT, the X-coordinate of the location where bits are to be placed
	6) DESTINATIONBOTTOM, the Y-coordinate of the location

where bits are to be placed
7) WIDTH, the width of the rectangle to copy
8) HEIGHT, the height of the rectangle to copy
9) SOURCETYPE,
10) OPERATION, the bitblt operation
11) TEXTURE, a texture handle
12) CLIPPINGREGION, a clipping region
13) CLIPPEDSOURCELEFT, left coordinate of the clipping region coordinate of the clipping region
15) SCALE, a scaling factor

Value: <user-defined>

7.9.10 Scaling While Bit-blitting

The **IMSCALEDBITBLT** method allows you to scale the data while bit-blitting it from one stream to another. It takes the form:

Method: <IMSCALEDBITBLT>
Arguments: 15
Arguments: 1) SOURCEBITMAP, the source bit map handle
2) SOURCELEFT, the X-coordinate of the bits to be copied
3) SOURCEBOTTOM, the Y-coordinate of the bits to be copied
4) STREAM, the destination stream handle
5) DESTINATIONLEFT, the X-coordinate of the location where bits are to be placed of the location where bits are to be placed
7) WIDTH, the width of the rectangle to copy
9) SOURCETYPE, the source object
10) OPERATION, the bit-blitting operation
11) TEXTURE, a texture handle
12) CLIPPINGREGION, a region specification
13) CLIPPEDSOURCELEFT, X-coordinate of lower left corner of the clipping region
14) CLIPPEDSOURCEBOTTOM, Y-coordinate of lower left corner of clipping region
15) SCALE, a scaling factor
Value: <user-defined>

Each pixel in SOURCEBITMAP is replicated SCALE times in the X and Y directions in the destination stream. Currently, SCALE must be an integer (e.g., satisfy FIXP). This method allows you to magnify (or reduce) an object when bit-blitting it from the source to the destination bitmap.

7.9.11 MOVING THE OBJECT

The **IMMOVETO** method allows you to move the cursor to the position (X,Y) within the specified stream. It takes the form:

Method: <IMMOVETO>
Arguments: 3
Arguments: 1) STREAM, a stream handle
2) X, an integer
3) Y, an integer
Value: The position (X,Y).

IMMOVETO is invoked by MOVETO. If a value is not supplied for it, a default method composed of calls to IMXPOSITION and IMYPOSITION is used.

7.9.12 DETERMINING STRING WIDTH

The method **IMSTRINGWIDTH** is used to determine the width of a string in units appropriate to the current stream. Because devices have different display characteristics, the string width must be dynamically computed when laying out a display. It takes the form:

Method:	<IMSTRINGWIDTH>
# Arguments:	3
Arguments:	1) STREAM, a stream handle
	2) STR, a string
	3) RDTBL, a read table handle
Value:	The width of the string.

IMSTRINGWIDTH is invoked by STRINGWIDTH. it uses the current font of the specified stream to calculate the width of the string in pixels. Consider IM.STRING.WIDTH, which calculates the string width for a generic stream:

```
(DEFINEQ
   (IM.STRING.WIDTH (STREAM STRING RDTBL)
   (PROG (STRING-WIDTH-BASE DD)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SETQ STRING-WIDTH-BASE
         (DISPLAYDATAPROP DD 'DDWidthscache))
      (RETURN
         (\STRINGWIDTH.GENERIC
               STRING
               STRING-WIDTH-BASE
               RDTBL
               (\GETWIDTH STRINGWIDTH-BASE (CHARCODE SPACE))
               )))
   ))
```

7.9.13 DETERMINING CHARACTER WIDTH

The **IMCHARWIDTH** method allows you to compute the width of a character in the units of the specified stream. It takes the form:

Method:	<IMCHARWDITH>
# Arguments:	2
Arguments:	1) STREAM, a stream handle
	2) CHARCODE, a character code
Value:	The width of the character.

Using the current font of the stream, IMCHARWIDTH will calculate the width of the character as it would be displayed in the stream in the appropriate units. Consider the function IM.CHARACTER.WIDTH, which is defined below.

```
(DEFINEQ
   (IM.CHARACTER.WIDTH (STREAM CHARACTER-CODE)
   (PROG (DD)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (RETURN
```

```
        (\GETWIDTH (DISPLAYDATAPROP DD 'DDWidthscache) CHARACTER-CODE))
))
```

Note that the character widths for the font currently in use are cached in the DISPLAYDATA structure for easy access.

7.9.13.1 Determining the Y-Component

The **IMCHARWIDTHY** method determines the Y-component of the width of the character specified as the argument. It takes the form:

Method:	<IMCHARWIDTHY>
# Arguments:	2
Arguments:	1) STREAM, a stream handle
	2) CHARCODE, a character code
Value:	The Y component of the width of the character.

Given the function for determining the character widths in Section 7.9.13, we can define the function IM.CHARACTER.WIDTHY to retrieve the Y-component as follows:

```
(DEFINEQ
   (IM.CHARACTER.WIDTHY (STREAM CHARACTER-CODE)
      (CADR (IM.CHARACTER.WIDTH STREAM CHARACTER-CODE))
   ))
```

7.9.14 DETERMINING THE BIT MAP SIZE

The **IMBITMAPSIZE** method determines the size that a specified bit map will be when bit-blitted to a destination stream. It takes the form:

Method:	<IMBITMAPSIZE>
# Arguments:	3
Arguments:	1) STREAM, a stream handle
	2) BITMAP, a bitmap handle
	3) DIMENSION, the dimension to compute
Value:	An integer or dotted pair.

IMBITMAPSIZE computes the size of the bit map in the units of the destination stream for the specified dimension. DIMENSION may take one of three values:

1. WIDTH
2. HEIGHT
3. NIL, whence a dotted pair of (WIDTH . HEIGHT) is returned

7.9.15 STARTING A NEW PAGE

The **IMNEWPAGE** method causes a new page to be started for the specified stream. It takes the form:

Method:	<IMNEWPAGE>
# Arguments:	1
Arguments:	1) STREAM, a stream handle
Value:	The position on the page representing the upper left corner.

If no value is supplied for this method, it defaults to the following expression:

```
(\OUTCHAR STREAM (CHARCODE ^L))
```

We can define the function IM.NEW.PAGE which will set a new page for the stream. First, IM.NEW.PAGE saves the current page and then calls the function IM.START.PAGE to setup the page parameters. These functions are defined as follows:

```
(DEFINEQ
   (IM.NEW.PAGE (STREAM)
   (PROG (DD)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SAVE.PAGE.BUFFER  (DISPLAYDATA DD 'DDDestination) STREAM)
      (IM.START.PAGE STREAM)
   )))

(DEFINEQ
   (IM.START.PAGE (STREAM)
   (PROG (DD CLIPPING-REGION PAGE-BUFFER)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SETQ CLIPPING-REGION (DISPLAYDATAPROP DD 'DDClippingRegion))
      (SETQ PAGE-BUFFER (DISPLAYDATAPROP DD 'DDDestination)))
      (\BLTSHADE.BITMAP  WHITESHADE PAGE-BUFFER NIL NIL NIL NIL 'REPLACE)
      (SET.DSP.POSITION  STREAM
                         (POINT (DISPLAYDATAPROP DD 'DDLeftMargin)
                                (fetch TOP of CLIPPING-REGION)))
   )))
```

IM.START.PAGE sets the contents of the page buffer bitmap to white. It then sets the current display stream position to the upper left hand corner as specified by the left margin and the top of the clipping region.

7.9.16 Starting a New Line

The **IMTERPRI** method causes a new line to be started in the image stream. It takes the form:

Method:	<IMTERPRI>
# Arguments:	1
Arguments:	1) STREAM, a stream handle
Value:	<user-defined>

If no value is supplied for this method, it defaults to the following expression:

```
(\OUTCHAR STREAM (CHARCODE EOL))
```

Usually, a function defined for this method will handle line spacing. Thus, based on a user-defined variable, the function will emit the appropriate number of EOL characters to ensure the proper spacing.

7.9.17 Resetting the Stream Position

The IMRESET method resets the X- and Y-coordinates of the image stream. It takes the form:

Method:	<IMRESET>
# Arguments:	1

Arguments: 1) STREAM, a stream handle
Value: NIL.

The function IM.RESET, which is defined below, resets the X- and Y-coordinates of the image stream to the upper left hand corner of the current page.

```
(DEFINEQ
    (IM.RESET (STREAM)
    (PROG (CLIPPING-REGION DD)
        (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
        (SETQ CLIPPING-REGION
            (DISPLAYDATAPROP DD 'DDClippingRegion))
        (SET.DSP.POSITION
              STREAM
              (POINT
                  (DISPLAYDATAPROP DD 'DDLeftMargin)
                  (fetch TOP of CLIPPING-REGION)))
        (BITBLT  NIL
              NIL
              NIL
              STREAM
              (fetch LEFT of CLIPPING-REGION)
              (fetch BOTTOM of CLIPPING-REGION)
              (fetch WIDTH of CLIPPING-REGION)
              (fetch HEIGHT of CLIPPING-REGION)
              'TEXTURE
              'REPLACE
              (DISPLAYDATAPROP DD 'DDTexture))
    )))
```

7.9.18 Setting the X and Y Positions

The IMXPOSITION and IMYPOSITION methods set the X and Y positions of the specified stream. They take the form:

Method: <IMXPOSITION>
<IMYPOSITION>
Arguments: 2
Arguments: 1) STREAM, a stream handle
2) POSITION, an X or Y position, respectively
Value: The X- or Y-coordinate, respectively.

Two functions, IM.XPOSITION and IM.YPOSITION, are provided for setting the X- or Y-coordinates, respectively, of the stream.

```
(DEFINEQ
    (IM.XPOSITION (STREAM XPOSITION)
    (PROG (DD XPOS)
        (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
        (SETQ XPOS (DISPLAYDATAPROP DD 'DDXposition))
        (if    (NULL XPOSITION)
         then
              NIL
         elseif
              (NUMBERP XPOSITION)
```

```
            then
                 (DISPLAYDATAPROP DD 'DDXposition XPOSITION)
            else
                (\ILLEGAL.ARG XPOSITION))
      (RETURN XPOS)
   )))

(DEFINEQ
   (IM.YPOSITION (STREAM YPOSITION)
   (PROG (DD YPOS)
       (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
       (SETQ YPOS (DISPLAYDATAPROP DD 'DDYposition))
       (if (NULL YPOSITION)
         then  NIL
         elseif (NUMBERP YPOSITION)
                then
                (DISPLAYDATAPROP DD 'DDXposition YPOSITION)
         else (\ILLEGAL.ARG YPOSITION))
      (RETURN YPOS)
   )))
```

7.9.19 Setting the Stream Font

The IMFONT method sets the current font of the specified stream. It takes the form:

Method:	<IMFONT>
# Arguments:	2
Arguments:	1) STREAM, a stream handle
	2) FONT, a font handle
Value:	The old font descriptor.

The function IM.FONT, which sets the font that the image stream will use to print characters, is defined below:

```
(DEFINEQ
   (IM.FONT (STREAM FONT)
   (PROG (DD OLDFONT)
       (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
       (SETQ OLDFONT (DISPLAYDATAPROP DD 'DDFONT))
       (if FONT
         then (OR (EQ FONT OLDFONT) (DISPLAYDATAPROP DD 'DDFONT FONT)))
       (RETURN OLDFONT)
   )))
```

Note that we only change the font if it is not equal to the old font.

7.9.20 SETTING THE LEFT AND RIGHT MARGINS

You may set the left and/or right margins of the image stream using the methods IMLEFTMARGIN and IMRIGHTMARGIN, respectively. They take the form:

Method:	IMLEFTMARGIN
	IMRIGHTMARGIN
# Arguments:	2

Arguments:	1) STREAM, an image stream handle
	2) MARGIN, an integer
Value:	The old margin value.

IMLEFTMARGIN sets the left margin of the image stream. The left margin is defined as the X-position in the stream to which the cursor is set when a new line is generated.

IMRIGHTMARGIN sets the right margin of the image stream. The right margin is defined as the maximum X-position in the image stream at which characters will be printed. Attempting to print characters beyond this position will cause a new line to be started.

The functions for setting the left and right margins, IM.LEFT.MARGIN and IM.RIGHT.MARGIN, are defined below:

```
(DEFINEQ   (IM.LEFT.MARGIN (STREAM XPOSITION)
   (PROG (DD LEFTMARGIN)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SETQ LEFTMARGIN (DISPLAYDATAPROP DD 'DDLeftMargin))
      (AND XPOSITION
           (if (AND (SMALLP XPOSITION) (IGREATERP XPOSITION -1))
            then (DISPLAYDATAPROP  DD  'DDLeftMargin XPOSITION)
            else  (\ILLEGAL.ARG XPOSITION)
      ))
      (RETURN LEFTMARGIN)
   )))

(DEFINEQ    (IM.RIGHT.MARGIN (STREAM XPOSITION)
   (PROG (DD RIGHTMARGIN)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SETQ RIGHTMARGIN (DISPLAYDATAPROP DD 'DDRightMargin))
      (if       (NULL XPOSITION)
       then    (RETURN RIGHTMARGIN)
       elseif  (AND (SMALLP XPOSITION)  (IGREATERP XPOSITION -1))
            then
            (PROGN
               (DISPLAYDATAPROP  DD 'DDRightMargin  XPOSITION)
               (\SFFIXLINELENGTH STREAM))
            else  (\ILLEGAL.ARG XPOSITION))
      (RETURN RIGHTMARGIN)
   )))
```

We test for a smallp integer as a bounds test on the value of the left or right margins. In setting the right margin, the line length may be affected and so we call the system function \SFFIXLINELENGTH to calculate a new line length.

7.9.21 Setting the Top and Bottom Margins

You may set the top and/or bottom margins of the image stream using the methods IMTOPMARGIN and IMBOTTOMMARGIN, respectively. They take the form:

Method:	IMTOPMARGIN
	IMBOTTOMMARGIN
# Arguments:	2
Arguments:	1) STREAM, an image stream handle
	2) MARGIN, an integer

Value:	The old margin value.

IMTOPMARGIN sets the top margin of the image stream. The top margin is defined as the Y-position of the at which the tops of characters will be printed after a new page is started.

IMBOTTOMMARGIN sets the bottom margin of the image stream. The bottom margin is defined as the maximum Y-position at which characters will be printed. Attempting to print characters beyond this position will cause a new page to be started.

7.9.22 SETTING THE LINE FEED DISTANCE

You may set the distance at which a new line will be positioned after a new page is started using the method IMLINEFEED:

Method:	IMLINEFEED
# Arguments:	2
Arguments:	1) STREAM, an image stream handle
	2) DELTA, an integer
Value:	The old line feed value.

The line feed distance is calculated by multiplying the height of the current font by DELTA. This generates a new Y-position from the top margin of the image stream. The function IM.LINE.FEED sets a new line feed value for the stream.

```
(DEFINEQ       (IM.LINE.FEED (STREAM DELTA)
   (PROG (DD LINEFEED)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SETQ LINEFEED (DISPLAYDATAPROP DD 'DDLINEFEED))
      (AND    DELTA
              (if      (NUMBERP DELTA)
               then    (DISPLAYDATAPROP  DD  'DDLINEFEED DELTA)
               else    (\ILLEGAL.ARG DELTA)
      ))
      (RETURN LINEFEED)
   )))
```

While we do not explicitly test for a SMALLP in this example, it may be a good idea to do so in order to prevent unusual results in the display.

7.9.23 Determining the Scale of the Display Medium

You may determine the scale of the current display medium using the method IMSCALE:

Method:	IMSCALE
# Arguments:	2
Arguments:	1) STREAM, an image stream handle
	2) SCALE, is ignored
Value:	The number of device points.

IMSCALE determines the number of device points per screen point for the current display medium. Typically, a screen point is approximately 1/72 of an inch (.0138 of an inch).

7.9.24 SETTING THE SPACE FACTOR

You may set the spacing between characters using the method **IMSPACEFACTOR**:

Method:	IMSPACEFACTOR
# Arguments:	2
Arguments:	1) STREAM, an image stream handle
	2) FACTOR, an integer
Value:	The old spacing factor.

The spacing factor is used to adjust the spacing between characters in an image stream (e.g., between words which are separated by one or more spaces). This allows you to adjust the justification of text. The default value is 1. Thus, if your current font defines the natural width of a space to be 18 units and you set the space factor to 2, all spaces will appear to be 36 units wide. The values returned by STRINGWIDTH and CHARWIDTH are affected by the space factor.

7.9.25 SETTING THE DEFAULT BITBLT OPERATION

You may set the default bit-blit operation using the method IMOPERATION:

Method:	IMOPERATION
# Arguments:	2
Arguments:	1) STREAM, an image stream handle
	2) OPERATION, an operation
Value:	The old default operation.

IMOPERATION sets the default bit-blit operation to OPERATION. The function IM.OPERATION, which is defined below, ensures that a valid operation is set as the new operation for the stream.

```
(DEFINEQ            (IM.OPERATION (STREAM OPERATION)
   (PROG (DD OLD-OPERATION)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SETQ OLD-OPERATION (DISPLAYDATAPROP DD 'DDOPERATION))
      (if    OPERATION
       then
         (PROGN
            (OR (MEMBER   OPERATION '(PAINT REPLACE INVERT ERASE))
                  (ERROR "ILLEGAL ARG" OPERATION))
            (DISPLAYDATAPROP DD 'DDOPERATION OPERATION)
            <make other adjustments>
      ))
      (RETURN OLD-OPERATION)
   )))
```

You may have to modify other system data structures as a result of changing the operation. Several examples have suggested that the PILOTBBT data structure (which is not defined in the IRM) may have to be modified.

7.9.26 SETTING A NEW CLIPPING REGION

You may set the clipping region of an image stream (analogous to the clipping region of a display stream) using the method **IMCLIPPINGREGION**:

Method:	IMCLIPPINGREGION
# Arguments:	2
Arguments:	1) STREAM, an image stream handle
	2) REGION, a region specification

Value: The old clipping region specification.

IMCLIPPINGREGION sets the clipping region for the image stream to the region sepcified by REGION. The function, IM.CLIPPING.REGION, which is defined below, sets a new clipping region for the stream.

```
(DEFINEQ      (IM.CLIPPING.REGION (STREAM REGION)
   (PROG (DD OLD-REGION)
      (SETQ DD (IMAGESTREAMPROP STREAM 'IMAGEDATA))
      (SETQ OLD-REGION (DISPLAYDATAPROP DD 'DDClippingRegion))
      (if REGION
       then
         (PROGN
            (OR (type? REGION REGION)  (ERROR REGION "is not a region"))
            (DISPLAYDATAPROP DD 'DDClippingRegion REGION)
            (\SFFIXCLIPPINGREGION DD)
      ))
      (RETURN OLD-REGION)
   )))
```

When changing the clipping region for an image stream, you must also be careful to ensure that other system data structures are updated as well. The system function \SFFIXCLIPPINGREGION appears to do this.

INDEX