INFORMATION TO USERS

The most advanced technology has been used to photograph and reproduce this manuscript from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A WYSIWYG literate programming system

Wu, Cheng-Shiung Jesse, Ph.D.

The Ohio State University, 1990
A WYSIWYG LITERATE PROGRAMMING SYSTEM

DISSERTATION

Presented in Partial Fulfillment of the Requirements for
the Degree Doctor of Philosophy in the Graduate
School of The Ohio State University

By
Cheng-Shiung Jesse Wu, M.S.

The Ohio State University
1990

Dissertation Committee:
E. Gurari, Advisor
S. Lai
M. Singhal

Approved by
Eitan M. Gurari, Advisor
Department of Computer and Information Science
In memory of my mother
and
To my father
ACKNOWLEDGMENTS

I express my sincere appreciation to Dr. Eitan Gurari for his guidance, insight and patience throughout the research. Especially during the period I suffered from deadly diseases, without his consideration and encouragement, it would be difficult for me to stand up. Thanks go to Department of Computer and Information Science, Graduate School for financial support.

"Even though I walk through the valley of the shadow of death,
I will fear no evil, for you are with me;
Your rod and Your staff, they comfort me."

Thanks to the LORD
VITA

May 1, 1952

1975

B.S., National Cheng-Kung University
Tainan, Taiwan, R.O.C.

1981

M.S., The Ohio State University

1983

M.S., The Ohio State University

FIELDS OF STUDY

Major field : Computer and Information Science
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>ACKNOWLEDGMENTS</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td>VITA</td>
<td>iv</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vii</td>
</tr>
</tbody>
</table>

## CHAPTER

<table>
<thead>
<tr>
<th>I. BACKGROUND</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Literate Programming</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Literate Programming Systems</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Batch-Oriented Systems</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Interactive Systems</td>
<td>4</td>
</tr>
<tr>
<td>1.5 Our System</td>
<td>5</td>
</tr>
<tr>
<td>II. THE ProWrite SYSTEM</td>
<td>7</td>
</tr>
<tr>
<td>2.1 Writing</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Boxes</td>
<td>9</td>
</tr>
<tr>
<td>III. MERIT OF WORK</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Programming with ProWrite</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Critique</td>
<td>18</td>
</tr>
</tbody>
</table>

## APPENDICES

<table>
<thead>
<tr>
<th>A. MANUAL</th>
<th>21</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Creating and Opening Files</td>
<td>21</td>
</tr>
<tr>
<td>A.2 Editing Files</td>
<td>21</td>
</tr>
<tr>
<td>A.3 Creating Boxes and Ports</td>
<td>22</td>
</tr>
<tr>
<td>A.4 Peeling Boxes</td>
<td>22</td>
</tr>
<tr>
<td>A.5 Working on Boxes</td>
<td>22</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>A.6</td>
<td>Working on Ports</td>
</tr>
<tr>
<td>A.7</td>
<td>Selecting Fonts for Program Code</td>
</tr>
<tr>
<td>A.8</td>
<td>Printing Files</td>
</tr>
<tr>
<td>A.9</td>
<td>Closing and Saving Files</td>
</tr>
<tr>
<td>B.1</td>
<td>General Comments</td>
</tr>
<tr>
<td>B.2</td>
<td>The Main Module</td>
</tr>
<tr>
<td></td>
<td><strong>BIBLIOGRAPHY</strong></td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A view of the screen</td>
<td>8</td>
</tr>
<tr>
<td>2. Pull down menus</td>
<td>9</td>
</tr>
<tr>
<td>3. A literate program</td>
<td>11</td>
</tr>
<tr>
<td>(a) Top level</td>
<td>11</td>
</tr>
<tr>
<td>(b) Second level</td>
<td>11</td>
</tr>
<tr>
<td>(c) Remaining levels</td>
<td>12</td>
</tr>
<tr>
<td>4. Usage of ports</td>
<td>14</td>
</tr>
<tr>
<td>(a) Leading parts</td>
<td>14</td>
</tr>
<tr>
<td>(b) Middle parts</td>
<td>15</td>
</tr>
<tr>
<td>(c) Trailing parts</td>
<td>16</td>
</tr>
<tr>
<td>5. View of boxes</td>
<td>30</td>
</tr>
<tr>
<td>(a) Target box</td>
<td>30</td>
</tr>
<tr>
<td>(b) The pointer curB1ck</td>
<td>31</td>
</tr>
<tr>
<td>(c) Dummy lines</td>
<td>32</td>
</tr>
<tr>
<td>6. Possible locations of the caret</td>
<td>33</td>
</tr>
<tr>
<td>7. Location of the caret within the window and the box</td>
<td>35</td>
</tr>
<tr>
<td>8. Location of the caret within the window and a title</td>
<td>37</td>
</tr>
<tr>
<td>9. Location of the caret within the window and the title of an opened port</td>
<td>38</td>
</tr>
<tr>
<td>10. Location of the caret within the window and the text of an opened port</td>
<td>40</td>
</tr>
<tr>
<td>11. Caret in a title that appears within the body of an opened port</td>
<td>41</td>
</tr>
<tr>
<td>12. Access to the ruler</td>
<td>44</td>
</tr>
</tbody>
</table>
CHAPTER I

BACKGROUND

Programs are sequences of commands which specify how computations should be carried out. They are written for computers which execute them blindly without paying attention to the objective of the programs and to the motivation for their designs. They are quite easy to execute because they consist of primitive commands, but for this same reason they are normally large in size and complex in structure.

Writing programs of large sizes and complex structures is a difficult task, because at any given time a human being can handle only a small amount of information. Programming languages simplify this task by introducing abstract commands tailored for the problems under consideration and by providing the users with features for defining their own abstractions. Such abstractions allow for programs that are expressed in more natural fashion and that are reduced in size and structural complexity. Yet, with all the advantages that such abstractions offer even the most advanced programming languages have their limitations with regard to the extent to which they can facilitate the task of programming.

By their nature, programming languages are suitable mostly for stating what
the programs should do and less so for stating why the programs do whatever they do the way that they do it. That is, they do not provide their users with acceptable medium for documenting design issues. This shortcoming is evident in the pluralization of poorly documented pieces of software, despite the wide recognition that documentation is a must. A recognition that brought much preaching to produce documentation but very little in environments that genuinely support such activities.

1.1 Literate Programming

With this perspective in mind, the discipline of literate programming [8] deserves a special attention because it calls to write programs for computers the way that we explain them to human beings. That is, it promotes the writing of programs that interleave commentary with code, rearrange the different parts of the code to orders that better appeal to human readers, and use abstractions presented in a natural language (and implemented elsewhere) as substitutions for pieces of code. Programs of this kind are called literate programs and they assume the appearance of documents that describe programs.

Literate programs provide us a channel for communicating our thoughts between the different phases of the programming process, and provide us with encouragement to clarify our thoughts when we write them down and to refine them when we review them. Moreover, they allow us to decompose and rearrange the code based on logical considerations, and to use enhanced visual expositions for highlighting logical units and structural relationships between the units.
1.2 Literate Programming Systems

Systems that provide the tools needed for practicing literate programming are called literate programming systems. Systems of this kind at the very least must offer the features that document preparation systems offer, should provide features for identifying fragments of code and relations between fragments of code, and should be equipped with the ability to extract the code from the documents. On the other hand, the more sophisticated literate programming systems should also assist their users in navigating around the code and should be able to interact with the agents that execute the programs.

Computers that are sufficiently powerful to run literate programming systems became available to the community of programmers only in recent years. Hence, it is not surprising that literate programming systems are new entries in the field, and that the currently available systems are few in number and lean in features.

1.3 Batch-Oriented Systems

The first literate programming system to appear was WEB [8]. In essence, WEB is a preprocessor that gets fragments of a Pascal [20] program and pieces of prose mixed together and identified with special markers. From such an input, called a source literate program, WEB generates two files: one file containing the Pascal program for an agent that executes the program, and a second file containing the literate program with formatting instructions for generating a viewing copy with TeX [7]. In addition, WEB produces cross referencing information and an index, to facilitate the navigation between the different fragments of code.

Since its appearance, variants of WEB have been written for other programming languages besides Pascal, for instance, [1, 3, 9, 14, 15, 16, 19]. In particu-
lar, [14] provides a version of WEB which facilitates the tailoring of the system to
different programming languages (but it requires some knowledge of the architec-
ture of WEB [16]). On the other hand, [4] introduced a small extension to \TeX
which provides a highly-portable language-independent system that can be easily
adjusted to meet different specifications.

The above literate programming systems are very useful systems. However,
they are suitable mainly for sophisticated programmers because they are quite
complex and costly to operate. Specifically, they expect their users to write and
modify their literate programs indirectly through source literate programs. More-
over, they require that their users will navigate around between a literate pro-
gram, a source literate program, and a computer program whenever errors arise
and changes are to be made. Further more, they increase the amount of time that
it takes for making changes in code, because the literate programming systems
have to be activated to propagate the changes from the source programs. The in-
crease in time is even more severe when the users work with interactive compilers
and interpreters, because the stand alone literate programming systems dictate a
batch-oriented mode of programming. (Yet, the increase in time for each cycle of
modifications is more than compensated for by the reduced number of cycles.)

1.4 Interactive Systems

In contrast to the above systems, BOXER [2] offers a programming environment
for unsophisticated users. It was not constructed with the intention to make it
a literate programming system. Instead, its intended objective was to provide an
environment for constructing electronical books that can be dynamically changed
by their readers. Yet, in some respects BOXER has the flavor of a futurist literate
Specifically, BOXER can be used to prepare a literate program on a bitmap screen in a what-you-see-is-what-you-get (WYSIWYG, for short) [17] mode. A literate program of BOXER contains text and boxes, where the boxes are rectangular regions with visible boundaries. The boxes may contain text or (LOGO [13]) code, and they can be organized into hierarchies that relate their contents. In addition, BOXER provides an interpreter for executing the code that is contained within the boxes. The user designates special boxes from which the interpreter gets its input, and designates special boxes for holding the output of the interpreter.

As a literate programming system, BOXER is simpler to work with than the previous systems because it communicates with its users at their level. That is, the users write their literate programs directly the way that they perceive them in their heads, and BOXER responds by working directly on these programs. Moreover, BOXER highlights the procedures, allows to hide their implementations, and supports hierarchical organization for the documents by allowing to enter and leave boxes whose dimensions can be dynamically changed.

However, BOXER is weaker than the batch-oriented systems because it does not provide mechanisms for fragmenting and abstracting code (except for the procedural abstractions that LOGO supports). Moreover, it provides only primitive features for the writing activities, and it is a complex system to change for meeting different requirements and for accommodating other programming languages.

1.5 Our System

The objective of our work [5] is to introduce a new literate programming system that improves on the currently available systems. The system is discussed in the
remaining parts of this dissertation, and for convenience we refer to our system by the name ProWrite.
CHAPTER II

THE ProWrite SYSTEM

The literate programming system ProWrite offers tools for writing two dimensional documents. The first section of this chapter describes how the system can be used for writing horizontally (that is, on the screen plane), and the second section describes how the system can be used for writing vertically.

2.1 Writing

A literate programming system can be viewed as a document preparation system that is extended with features for defining programs. Our system uses a hierarchical variation of MacWrite [11] for the kernel that provides the features that are needed for the writing process.

A user of ProWrite writes on a bitmap screen (see Figure 1). The writing can be freely done with no restriction on form and content in a WYSIWYG mode of communication. The system distinguishes between code and prose by the fonts that they are written in.

A pull down menu bar is provided for controlling the system (see Figure 2). Many of the commands are quite standard, and they are similar in their appearance
In literate programs, prose and code can be mixed together.

```
program maximum;
  var x, y, f, max: integer;
begin
  read(x,y);
  f := (2 * (y-x) + 3) \div (2 * (y-x) + 1);
  f := f*f+1;
  f := f \div 2;
  f := f*(y-x);
  max := x+f;
  write(max)
end.
```

This literate program uses plain Geneva font of size 12 points for prose, and plain and bold Monaco fonts of size 9 points for code. (Note that all the characters in each monaco font have the same width).

Figure 1 A view of the screen.

to those provided by other systems. They include commands for opening and closing files, commands for editing and formatting text, and so on. On the other hand, some commands are unique for ProWrite because they are tailored to take care of the unique features of the system.

From the point of view of ProWrite, the document that a user writes is a literate program. The code can be extracted from the literate program with the command Save Code... that is listed in the File menu (and then be introduced to any compiler or interpreter). The commands in the Code menu provide the means
for choosing the fonts that identify code. A diamond appears near the header of the Code menu, when any of these fonts is active.
2.2 Boxes

The classification of text into prose and code allows to fragment code for providing commentary that is physically adjacent to the fragments of code. However, it does not allow to rearrange code fragments to orders that differ from those that are imposed by the programming language. ProWrite generalizes the notions of boxes and ports of BOXER to support such mode of writing.

A box in ProWrite consists of two parts: a title and a body. A box is defined by choosing a piece of text (with the mouse) and either making it a title of a box with empty body or making it a body of a box with empty title. The first case is achieved with the command Box Title from the Box menu, whereas the second case is achieved with the command Box Body.

ProWrite encloses each title of a box within a rectangle (see Figure 3(a)). The title of a box can be modified like any other text in a literate program, but its content is regarded as prose no matter what fonts it uses. By pointing to a title of a box and then issuing the command Enter Box, the window of ProWrite moves to the expansion of the box.

A window into an expansion of a box, exhibits the title and the body of the box with a line separating the two parts (see Figure 3(b)). The title and the body can be edited like any other text in the literate program. Moreover, bodies of boxes can include other boxes to provide for hierarchy of boxes (see Figure 3(c)). The command Exit Box reverses the effect of the command Enter Box.

Boxes can be viewed and modified anywhere within the literate programs by inserting ports at the desired locations (see Figure 4). A port can either show just the title of a box or the whole expansion of the box.
The following program reads in two integer values, and writes out the larger value.

```pascal
program maximum;
var x, y, f, max: integer;
begin
  read(x, y);
  if y < x then f := 0
  else f := y - x;
  max := x + f;
  write(max);
end.
```

The program uses only the operations of plus (+), minus (-), multiplication (*), and division (div).

The desired value of f can be computed in the following manner.

```pascal
f := 0 if y < x
  1 if y > x
f := f + (y - x);
```

Note that,

a. Within the nested box, f can take any value when y = x.

b. The nested box acts as precondition, and the title of the current box acts as post condition, for the Pascal code. (The same holds in the higher levels of nesting in this program.)

Figure 3 A literate program.

(a) Top level.

(b) Second level.
In this level, the Pascal code is empty.

In this level, the precondition is empty.

(c)

Figure 3- cont. A literate program.

(c) Remaining levels.
ProWrite uses a depth first traversal over the content of a two dimensional literate program, when it extracts the code of the program. On the other hand, when ProWrite prints the whole literate program it linearizes the literate program in accordance to a depth first order of the boxes.
program sort;
storage for set
begin
  read set
  sort set
  write set
end.

storage for set
const n=100;
var A: array [1..n] of integer;
The content of this box is defined incrementally as the need for more declarations arise.

read set
for i:= 1 to n do read(A[i]);
The variable "i" is declared here.

storage for set
const n=100;
var A: array [1..n] of integer;
i: integer;
The content of this box is defined incrementally as the need for more declarations arise.

Figure 4 Usage of ports.
(a) Leading parts.
for i := 1 to n-1 do
for j := i to n do
if a[i] > a[j] then

We declare here the variable "j".

storage for set
const n = 100;

var A: array [1..n] of integer;
i, j: integer;

The content of this box is defined incrementally as the need for more declarations arise.

***

begin
end;

Here we add the variable "save" to the declarations.

storage for set
const n = 100;

var A: array [1..n] of integer;
i, j, save: integer;

The content of this box is defined incrementally as the need for more declarations arise.

***

Figure 4-cont. Usage of ports.

(b) Middle parts.
write set

for $i := 1$ to $n$ do write($A[i]$);

The (closed) port is inserted here to provide a fast access into the variables that are declared for the current box.

Figure 4-cont. Usage of ports.

(c) Trailing parts.
CHAPTER III

MERIT OF WORK

In its current stage, ProWrite is just a prototype of the system that we aspire for. It captures the spirit of our vision for a WYSIWYG literate programming system, but it still has much room for improvement.

3.1 Programming with ProWrite

Literate programming systems offer features for identifying the different components of the programs, for highlighting the relations that exist between the components, and for explaining the components and the relations. ProWrite is no exception in this respect, but it does so with a medium that is more natural for human beings.

When code and prose are written with ProWrite they are displayed immediately on the screen with their designated content and form. Such an instant feedback lets the users review and modify their writings while their thoughts are still fresh in their minds. Moreover, the WYSIWYG mode of writing lets them work directly with concrete copies of the literate programs that they carry in their minds.
When boxes are used within programs, their titles are treated as abstract pieces of code and their bodies are viewed as realizations for the abstract pieces of code. Hence, the ability to easily move (with the commands Enter Box and Exit Box) between the titles of the boxes and their corresponding bodies provides for a natural two-dimensional representation of programs. Each level of a program uses some combination of actual and abstract code, and the deeper we reach into the programs the more refined the code gets. The ports are used for generalizing different views of the programs by listing together code fragments that are logically (but not necessarily syntactically) related.

Conceptually, the command Box Title provides support for the methodology of top-down programming, whereas the command Box Body provides support for the methodology of bottom-up programming. Yet, ProWrite does not impose any methodology of programming. It just encourages its users to tailor their own approaches based on the above methodologies.

In fact, ProWrite does not impose any restriction on the programs being written with the system. It just encourages the philosophy that calls to work with documents that describe code instead of working with pure code, and it does so by providing tools that help pursue such an approach. Hence, the system benefits its users the most when they work within the philosophy that the system promotes.

The tools that ProWrite provides deal only with enhancements that can make the presentations of the programs more accessible to human beings. The tools do not relate to the content of the programs.

### 3.2 Critique

Since ProWrite relates only to style and not to content, it can be used as an
universal system for writing literate programs in different programming languages. In this respect, ProWrite is similar to ProTeX and differs from BOXER and WEB. The obvious advantage of having a universal system has also its drawback in the lack of language dependent features, like the automatic formatting of code and the automatic indexing of identifiers that WEB provides.

The pure WYSIWYG environment of writing that ProWrite currently supports, makes it easier to use ProWrite than the batch-oriented systems. However, such an environment deprives sophisticated users from the ability to globally control the style of their documents. A provision that the batch-oriented systems support through macro facilities, and that to some extent should be incorporated also into systems like ProWrite.

The provision for choosing different fonts for text that is identified as code, has the traditional benefit of highlighting different types of code. For instance, such is the case with the common use of boldface type for highlighting keywords in programs. However, here font choosing introduces also the extra benefit of providing an easy way for converting code into prose and vice versa. Such a provision is in particular useful for defining different classes of code that can be easily activated and deactivated as the needs arise (for instance, debugging classes of code). This mechanism is similar in power to the use of macros for such a purpose in the batch-oriented systems. BOXER lacks such a provision because it identifies code by location.

Boxes and ports give the users of ProWrite a natural way for defining abstract fragments of code and accessing their implementations. From the accessibility and visualization point of views these features are similar to those provided by BOXER, but conceptually they are more powerful in ProWrite because in BOXER they can
be used only for abstractions that are supported by the programming languages. On the other hand, in the current version of ProWrite the boxes and ports are conceptually weaker than those allowed in WEB and its descendants because they are not equipped with features for exhibiting incremental additions of code to their implementations.

The importance of figures in literate programs [12] (see also Appendix B) provided us with the motivation to start incorporating drawing facilities similar to those that exist in MacDraw [10] into ProWrite. In BOXER, and to some extent also in the batch-oriented systems, the users can program figures into their documents. Moreover, in the latest systems the users are also provided with features for importing figures that have been prepared on other systems [7].

Finally, in its current form, ProWrite is a prototype which calls for a major additional investment. Its implementation is quite poor, largely because of frequent changes to the specifications of the system. Moreover, it lacks sophisticated editing features [18], and it has not been integrated yet into computational environments that provide the tools for compiling and interpreting literate programs directly [2] and for automatically generating additional views. Yet, we believe that with reasonable amount of investment we can considerably improved the implementation of ProWrite, and that already in its current form it can serve as an inspiration for a new family of literate programming systems.
APPENDIX A

MANUAL

The literate programming system ProWrite is a hierarchical variant of MacWrite. If one is familiar with MacWrite, he will have no trouble getting started.

A.1 Creating and Opening Files

When you start to use ProWrite, the system provides an edit window without title for you to start typing.

Closing an Editing File

To close an editing file, choose Close from the File menu. The edit window will disappear, ready for you either to create a new file or to open an existing file.

Creating a New File

To create a new file, choose New from the File menu. An edit window without title will appear, ready for you to start typing.

Opening an Existing File

To open an existing file, choose Open... from the File menu. A standard file dialog lets you select files to open.

A.2 Editing Files

The ProWrite editor uses standard Macintosh editing techniques the way you are used to. Cut, Copy, Paste, Insert Ruler, and Show Rulers all work the way
they do in other applications. The Font, Style and Size are also similar to other applications.

A.3 Creating Boxes and Ports

ProWrite uses Box to form a hierarchical structure. A box may have nested boxes. Each box in ProWrite consists of two parts: a title and a body, and it appears as a rectangle enclosing the title of the box. A port is similar to a box except that it has double bars on either side of the port.

Creating a Box

To create a box, choose Box Title from the Box menu. A rectangle will be created. If a piece of text is highlighted before choosing Box Title from the Box menu, the highlighted part will be the title of the box and is enclosed by the rectangle. Otherwise, only a rectangle with empty title will be created.

Users also can create a box by choosing Box Body from the Box menu, only if a piece of text is highlighted. The highlighted part will be the body of the box, and the box has an empty title. So, a rectangle with empty title will be created.

Creating a Port

To create a port, you need to highlight a box first, then choose Get Port from the Box menu, then choose Insert Port from the Box menu. A rectangle with double bars on either side will be created. Both the port, just created, and the box have the same title and body.

A.4 Peeling Boxes

To destroy the hierarchy, ProWrite is able to extract the body of a box from the box and to insert the body to the upper level of text at the place where the box is located.

Peeling a Box

To peel a box, first choose a box by clicking mouse in the box or highlighting the box, then choose UnBox from the Box menu. The rectangle of the box will
be removed, and the body of the box will be inserted in the text at where the box is located.

A.5 Working on Boxes

Users may view a box as a folder in Macintosh file system. In order to work in a box users have to get into the box as users open a folder. After users enter a box, they can edit both the title and the body of the box as they do for the top level of a file. Users can return to an upper level by exiting a box.

Entering a Box

To enter a box, first choose a box by clicking mouse in a box, a port, a highlighted box, a highlighted port, or an opened port, then choose Enter Box from the Box menu. The text of the current level will disappear, and the title and the body, separating by a line, of the box will be shown on the screen.

Exiting a Box

To exit a box and return to the previous level, choose Exit Box from the Box menu. The text of the box will disappear and the text of the upper level will be shown on the screen.

Moving a Box

To move a box, cut a piece of text containing a box. Then choose any place users like and paste it. After a box is cut, all related ports will be out of sight. Ports will re-appear after the box is pasted to the text. If users repeat to do paste operations again, all the ports in the clip board will be no longer ports of some boxes. All ports become boxes when they are inserted in the text of a literate program.

A.6 Working on Ports

A box and a port are like a coin of two sides. A port has two types: close and open. A close port is a rectangle with double bars on either side, which enclosing the title of a box. An open port include the title and the body of a box, which is
By entering a box, users can view only one box. In order to view several boxes on one level at a time, ProWrite supports users to insert a port of a box at any level and at any place. Users are able to view the whole box, including the title and the body, through opening the port of the box. Users can also shut an open port.

**Opening a Port**

To open a port, first choose a port, then choose Open Port from the Box menu. The port will be replaced by an open port, which includes the title and the body of the box related to the port.

**Closing a Port**

To close a port, first click the mouse at any place in an open port, then choose Close Port from the Box menu.

### A.7 Selecting Fonts for Program Code

ProWrite supports users to select fonts for either document or code. Those fonts listed in the Code menu are code fonts. The rest are document fonts.

**Choosing Code Fonts**

To choose fonts as code fonts, users simply choose the fonts they like and then choose Add from the Code menu.

**Removing Code Fonts**

To delete a code font, first choose a font item in the Code menu, then choose Delete from the Code menu.

**Selecting Code Fonts for Text**

To choose a code font as current font for editing, first choose a font item in the Code menu, then choose Select from the Code menu.
A.8 Printing Files

ProWrite supports users to print a literate program of hierarchical structure in two ways, one is to print all levels of the text, the other is to print current level of the text of a box.

Printing All Levels

To print all levels of a literate program, choose Print All Levels from the File menu.

Printing a Box

To print one level of the text of a box, choose Print This Level... from the File menu. Users will see the standard printing dialog for the printer they are using.

A.9 Closing and Saving Files

ProWrite is able to extract program code from a literate program. When only the program code is saved, ProWrite will extract all the texts in the bodies of boxes associated with the fonts listed in the Code menu and add comments with "{box title...}" before the program code of the body of a box and "{...box title}" at the end.

Saving A Literate Program

To save a literate program, choose Save As... from the File menu. A standard save dialog asks users for the new name and then saves the contents of a literate program.

Saving Program Code

To save only a program code without any comment except the title of a box, choose Save Code... from the File menu. A standard save dialog asks users to enter a new name and then saves the program code of a literate program.
APPENDIX B

LITERATE VIEW
OF THE CODE

We are currently documenting and loading the implementation of ProWrite into ProTeX [4] in preparation for the next revision of our system. This appendix lists the main module in the implementation, with the objective of giving some insight on how a WYSIWYG system for literate programming can be constructed. However, the reader should be aware that the appendix consists just of a literate piece of software written for programmers rather than for publication.

B.1 General Comments

The following information is helpful for understanding the program.

a. The names of procedures and functions that belong to the system start with two or more uppercase characters. The names of procedures and functions that do not belong to the system start with a single uppercase character. All the other identifiers start with lowercase characters.

b. The character @ before a procedure name stands for a pointer to the procedure name.

c. A pointer is an address to a memory that can not be relocated. An handle is a pointer to a secondary pointer, where the secondary pointer can hold the address of a block of memory that can be relocated. Handles allow the operating system to compact memory while modifying the addresses in the secondary pointers ([6] I-75).
B.2 The Main Module

```
program Editing;
usesGlobalsLSP, SetUpMenuLSP, GetFileLSP, ProcessTopLevelLSP,
    ProcessTextLSP, ProcessGfLSP;
<< Definition of variables >>
<< Procedures and Functions for Main Program >>
begin
<< Action to be taken when users run out of memory >>
<< Initialize the screen and the document file >>
repeat << event >> until done; {= command quit}
<< Restore system menu >>
end.
```

Different events can come up during the program execution. Some events are taken care by the operating system (hence, the procedure SYSTEMTask), and some are taken care by ProWrite. Note,

a. Events: pressed button, released button, pressed key.

b. Not an event: moving a mouse.

c. The operating system records the events (including location and time) in a special queue (and ignores them when the queue becomes full—around 20 events).

d. By setting a menu, we implicitly ask the system to raise a flag whenever the corresponding events arise.

```
<< event >>
SYSTEMTask;
{ Wait for event }
if FRONTWindow = editWindow then HandleIBeamFlashAndGetEventMain(curWdStorage)
else GetSystemEvent;
{ Process event }
if curWdType = gfWdType then ProcessGfEvent(curWdStorage)
else ProcessTextEvent(curWdStorage)
```

''Wait for event'':

a. There are several system windows (which belong to the apple menu — e.g., chooser, clock), whereas ProWrite uses a single window characterized by the record editWindow.

b. ProWrite assumes that at each instance its window shows the title and body of a box, and that the box at the root has empty title. curWdStorage points
to the box currently shown in the window. (curBoxStorage will make a better name than curWdStorage.)

c. If the active (=front) window on the screen belongs to ProWrite then the procedure HandleIBEamFlashAndGetEventMain waits for an event, otherwise GetSystemEvent waits for an event. Both procedures pass the information about the event that they detect with the global record event.

d. The reference to I-BEAM in the name HandleIBEamFlashAndGetEventMain and below is misleading. The reference should be to the insertion point shown by the caret (=vertical bar, i.e., the location of insertion within the box).

e. The procedure SYSTEMTask is responsible for handling system tasks that arise in the meantime (it is recommended that the procedure will be invoked at least 60 times per second, [6] I-442).

‘Process event’: 

a. Checks whether the event is within graph box or regular box (currently only the second case has an implementation).

B.2.1 Running Out of Memory

The following code prompts the operating system to call the function MyGrowZone when the user runs out of memory (see [6] II-43).

```pascal
  SETGrowZone(OMyGrowZone);
  safetyHandle := NEWhandle(1000);
```

The function notifies the user about the problem (the saving of the file is not implemented yet).

```pascal
  MyErrorAlert('Out of Memory. Click "OK" to save files and quit');
  EXITtoShell
end;
```

Memory is reserved for the above function with the following code. The function claims the with the system procedure DISPOSHandle.

```pascal
  safetyHandle := NEWhandle(1000);
```
B.2.2 Initialization of Screen and Document File

```
:::  
>> Initialize the screen and the document file <<
SetUpMenu;  { See SetUpMenuLSP --- better called set-up-screen }
SetANewFile;  { See GetAFileLSP --- opens a new file to record user's
document }
UNLoadSeg(SetUpMenu);

:::
```

The two procedures are compiled into a single segment. UNLoadSeg releases
the memory that the segment uses. Note,
a. A segment always occupies 32K of memory, no matter what fraction of this
memory the code requires.
b. UNLoadSeg does not care which of the modules in the segment is referenced.

B.2.3 Waiting for An Event (Front Window Belongs to ProWrite)

```
:::  
>> Procedures and Functions for Main Program <<
<< Utilities for handling the cursor >>
<< Utilities for calculating location+size of caret >>

procedure HandleIBeamFlashAndGetEventMain (curWdStorage: udPtr);
<< Variables for showing blinking caret >>
begin
  if curWdType = gfWdType then  { graph box }
  else begin {text box}
    with curWdStorage^ do
    with textPart^ do
      begin
        << Calculate position+size of caret in window >>
        << Wait for an event + show blinking caret + show mouse
        location >>
        end;
      end;
  end;
end;

:::
```

From the point of view of ProWrite a literate program is a hierarchy of boxes,
where each box has a title and a body with the top level having an empty title. At
a given instant, ProWrite uses its window to display the active box (or just part
of the box, if it is too large). The variable `curWdStore` points to the record of the active box (see Figure 5—"Wd" is misleading, "Box" would have been more appropriate). The caret can be invisible (e.g., within an invisible ruler) or visible. The location of the caret in the window is computed from the information that `curWdStore` provides.

`LineType` specifies the type of the line that contains the caret (a ruler line, a text line, an expanded-port line, or a "dummy" line containing a graphical-line above or below a title of a box).

If the caret should be invisible (true-part) then it is displayed outside the window `(SETrect(iBeamRect, left, up, right, down) — the 'if' should be moved into the first case because it applies only there). Otherwise, we proceed according to the case (see Figure 6).

```plaintext
calculate position+size of caret in window if (not showRuler) and (curScreenLine-.lineType = rulerLineType) then
  SETrect(iBeamRect, -1, -2, 0, -1)
```
else
case curIBeamPart of
  titleArea, textArea:
    SetIBeamAtOpenWdMain(textPart);
  wdTitleArea, portTitleArea:
    SetIBeamAtWdMain(textPart);
  titlelnExpandPortArea:

Figure 5-cont. View of boxes.
(b) The pointer curBlck.
The procedure INVERTRect(iBeamRect) inverts the caret, and the procedure HandleEditCursorMain(curWdStorage) shows the location of the mouse. The user can use the control panel to select the rate of blinking (recorded in the variable GETCaretTime).

```
systemTask;
setiBeamAtTextInExpandPortMain(textPart);
setiBeamAtTitleInExpandPortMain(textPart);
if (i mod GETCaretTime) = 0 then
begin
   ihnVerTRect(iBeamRect);
end;
```

Figure 5-cont. View of boxes.
(c) Dummy lines.
samecolor := not Samecolor;
i := 1;
end;
end;
if sameColor then INVERTRect(iBeamRect);

\[ \begin{array}{l}
\text{Variables for showing blinking caret} \\
\text{\var i: integer; sameColor: boolean}
\end{array} \]

The Boolean variable SameColor has a true value when the caret is invisible.

**B.2.3.1 Showing the Cursor Position**

The screen contains a menu bar and a window. The window is made up of the document window and the scroll bar. The cursor has the shape of I-Beam within the document window, and the shape of an arrow elsewhere.
procedure HandleEditCursorMain (curWdStorage: wdPtr);
var
  r: rect;
  mouseLoc: point;
begin
  GETMouse(mouseLoc);
  SETRect(r, 0, 0, 495, 317);
  if PTInRect(mouseLoc, r) then
    begin {Show document cursor}
      if not inWindow then
        << System cursor to document cursor >>
      else << Document cursor to document cursor >>
      end else { Show system cursor }
    begin
      INITCursor;
      inWindow := false;
      inRulerWindow := false
    end;
  end;

  <<<

  SETRect(r, left, up, right, down) sets the variable r to overlap the doc-
  ument window.
  inWindow remember whether the cursor is within the document window or not.
  Similarly, inRulerWindow remembers whether the cursor is within the ruler or not.

  << System cursor to document cursor <<

  begin
    inWindow := true;
    if InRulerRegionMain(curWdStorage, mouseLoc) then
      inRulerWindow := true; { Cursor is already an arrow }
    else
      SETCursor(iBeam"")
  end;

  <<<

  << Document cursor to document cursor <<

  if InRulerRegionMain(curWdStorage, mouseLoc) then begin
    if not inRulerWindow then { Enter ruler }
      begin
        INITCursor; inRulerWindow := true
      end
  end else if inRulerWindow then { Leave ruler }
    begin
      SETCursor(iBeam""); inRulerWindow := false;
    end

  <<<
Figure 7 Location of the caret within the window and the box.

B.2.4 Approximation for the Location of the Caret

Utilities for calculating location+size of caret

More utilities for handling the caret

Adjust Caret Location and Find Its Size

Approximation for Caret Location

All the utilities in this subsection get a value for their formal parameter thisTextStorage from the actual parameter curWdStorage^.textPart (see Figure 5).

B.2.4.1 Caret for TitleArea and TextArea

Figure 7 explains the calculations below.

procedure SetiBeamAtOpenWdMain (thisTextStorage: textRecordPtr);
var tempPoint: point;
begin
with thisTextStorage^ do
begin
  tempPoint.h := iBeamLoc.h;
  tempPoint.v := iBeamLoc.v + absoluteTopLineV;
end;
tempPoint gives an approximation to the location of the caret within the coordinates of the document window. SetiBeamRectMain makes the necessary adjustments (so that the caret will not fall between lines when double spacing is used), and finds the size of the caret.

B.2.4.2 Caret for wdTitleArea and portTitleArea

Figure 8 illustrates the computations below.

Note that we have the mapping

\[ \text{ConvertWdCoorToScreenCoorMain : (tlh, tlv) \rightarrow (innerTLH, innerTLV)} \]

We start with location "a" (given by windowTLCorner in the coordinate system of the current box) and go to location "b" (given by (innerTLH, innerTLV) in the coordinate system of the window).

The ruler for the title of the embedded box is invisible within the current box.

B.2.4.3 Caret for titleInExpandPortArea

The transformation here is similar to the previous one. We go from location "a"
Figure 8 Location of the caret within the window and a title.

(given by windowTLCorner in the coordinate system of the current box) to location “b” (given by (titleTLcornerH, titleTLcornerV) in the coordinate system of
Figure 9 Location of the caret within the window and the title of an opened port.

However, location “a” and location “b” differ somewhat from those that we had earlier Figure 9). Specifically,

Location “a” — Before it was at the top-left corner of the text of the title. Now it is in the top-left corner of the port. (The two locations are at constant distance from one another. Hence, the above conceptual discrepancy should be corrected (in favor of the second option?).)

Location “b” — In both cases the location is at the top-left corner of the embedded box. However, now the title part and its ruler are shifted rightward by
the amount windowExtraH (= the width taken by the two vertical lines to the
left of the title).

>> Approximation for Caret Location <<
 procedure SetiBeamAtTitleInExpandPortMain (thisTextStorage:
textRecordPtr);
 var
tlh, tlv: integer;
tempPoint: point;
innerTextStorage: textRecordPtr;
titleTLcornerH, titleTLcornerV: integer;
begin
with thisTextStorage* do
begin
 tlh := windowTLCorner.h;
 tlv := windowTLCorner.v + absoluteTopLineV;
innerTextStorage := curBlck*.newWd*.textPart;
with innerTextStorage* do
begin
 ConvertExpPortTitleToScreenCoorMain(innerTextStorage,
 tlh, tlv, titleTLcornerH, titleTLcornerV);
 tempPoint.h := titleTLcornerH + iBeamLoc.h;
 tempPoint.v := titleTLcornerV + iBeamLoc.v;
 SetiBeamRectMain(tempPoint, curScreenLine, iBeamRect)
end
end;
end;

B.2.4.4 Caret for textInExpandPortArea

See Figure 10 and Figure 8.

>> Approximation for Caret Location <<
 procedure SetiBeamAtTextInExpandPortMain (thisTextStorage:
textRecordPtr);
 var
tlh, tlv, innerTLcornerH, innerTLcornerV: integer;
tempPoint: point;
innerTextStorage: textRecordPtr;
begin
with thisTextStorage* do
begin
 tlh := windowTLCorner.h;
 tlv := windowTLCorner.v + absoluteTopLineV;
innerTextStorage := curBlck*.newWd*.textPart;
ConvertExpPortCoorToScreenCoorMain(innerTextStorage,
 tlh, tlv, titleTLcornerH, titleTLcornerV);
with innerTextStorage* do
begin
 if (not showRuler)
 and (curScreenLine^.lineType = rulerLineType) then
**Figure 10** Location of the caret within the window and the text of an opened port.

```plaintext
SETrect(iBeamRect, -1, -2, 0, -1) {hide the cursor}
else
begin
    tempPoint.h := innerTLCornerH + iBeamLoc.h;
    tempPoint.v := innerTLCornerV + iBeamLoc.v;
    SetiBeamRectMain(tempPoint, curScreenLine, iBeamRect)
end
end

end
end;
```
B.2.4.5 Caret for \texttt{wdTitleInExpandPortArea}

See Figure 11.

\begin{verbatim}
procedure SetiBeamAtTitleWdInExpandPortMain (thisTextStorage: textRecordPtr);
var
tempPoint: point;
innerTextStorage, innerMostTextStorage: textRecordPtr;
\texttt{tlh, tlv, innerTLCornerH, innerTLCornerV,}
innerMostTLCornerH, innerMostTLCornerV: integer;
begin
with thisTextStorage\^ do
begin
\end{verbatim}
innerTextStorage := curBlck^.newWd^.textPart;
tlh := windowTLCorner.h;
tlv := windowTLCorner.v + absoluteTopLineV;
ConvertExpPortCoorToScreenCoorMain(innerTextStorage, tlh, tlv, innerTLcornerH, innerTLcornerV);
with innerTextStorage do
begin
innerMostTextStorage := curBlck^.newWd^.textPart;
innerTLcornerH := innerTLcornerH + windowTLcorner.h;
innerTLcornerV := innerTLcornerV + windowTLcorner.v
end;
ConvertWdCoorToScreenCoorMain(innerMostTextStorage,
innerTLcornerH, innerTLcornerV,
innerMostTLcornerH, innerMostTLcornerV);
with innerMostTextStorage do
begin
tempPoint.h := innerMostTLcornerH + iBeamLoc.h;
tempPoint.v := innerMostTLcornerV + iBeamLoc.v;
SetiBeamRectMain(tempPoint, curScreenLine, iBeamRect)
end;
end;

B.2.5 Adjusting the Location of the Caret and Finding its Size

SetiBeamRectMain(tempPoint, curScreenLine, iBeamRect) is the manner in which the following procedure is called. caretLoc holds the approximate location of the caret, and caretRect is the caret rectangular.

```pascal
procedure SetcaretRectMain (caretLoc: point;
thisScreenLine: lineRecordPtr;
var caretRect: rect);
var
thisRulerStorage: rulerRecordPtr;
v: integer;
begin
thisRulerStorage := FindRulerByScreenLineMain(thisScreenLine);
with thisRulerStorage do
begin
if thisScreenLine^.lineType = textLineType then
begin
if spaceType = doubleSpace then
v := thisScreenLine^.height
else v := 0;
end else v := 0;
end;
with thisScreenLine^.lineFontInfo do
SETRect(caretRect, caretLoc.h, v + caretLoc.v,
```
FindRulerByScreenLineMain finds the ruler that precedes the caret (the case that the caret is inside a ruler has already been treated). The variable \( v \) holds the adjustment for the location of the caret. `SETRect` sets up the size of the rectangle that represents the caret. `ascent` and `descent` are measured from the base line.

### B.2.6 More Utilities for Caret

More utilities for handling the caret:

- `function FindRulerByScreenLineMain ([ ] : lineRecordPtr): rulerRecordPtr;`  
- `function HeightOfThisLineMain ([ ] : boolean; [ ] : rulerRecordPtr; [ ] : lineRecordPtr): integer;`  
- `function InExpandPortRulerRegionMain([ ] : lineRecordPtr; [ ] : integer; [ ] : point): boolean;`  
- `function InRulerRegionMain([ ] : wchar; [ ] : integer; [ ] : point): boolean;`  
- `function RulerLeftBoundaryMain([ ] : wchar);`  
- `procedure ConvertWdCoordToScreenCoordMain([ ] : textRecordPtr; [ ] : integer; [ ] : integer);`  
- `procedure ConvertExpPortTitleToScreenCoordMain([ ] : textRecordPtr; [ ] : integer; [ ] : integer);`  
- `procedure ConvertExpPortCoordToScreenCoordMain([ ] : textRecordPtr; [ ] : integer; var [ ] : integer);`  

### B.2.6.1 Finding the Ruler for The Current Line

See Figure 12.

More utilities for handling the caret:

- `function FindRulerByScreenLineMain ([ ] : lineRecordPtr): rulerRecordPtr;`  
- `function FindRulerByScreenLineMain (thisScreenLine: lineRecordPtr): rulerRecordPtr;`  
- `var stop: boolean;`  
- `tempScreenLine: lineRecordPtr;`  
- `thisRulerStorage: rulerRecordPtr;`  

begin
  thisRulerStorage := nil;
  if thisScreenLine <> nil then
    begin
      tempScreenLine := thisScreenLine;
      stop := false;
      while (tempScreenLine <> nil) and (not stop) do
        begin
          if tempScreenLine^.lineType = rulerLineType then
            begin
              stop := true;
              thisRulerStorage := tempScreenLine^.startBlck^.newWd^.rulerPart
            end
          end
        end
    end
end;
Figure 12 Access to the ruler.

```plaintext
end
else tempScreenLine := tempScreenLine^.up;
end;
end;
FindRulerByScreenLineMain := thisRulerStorage
end;

B.2.6.2 Finding the Height of Lines

function HeightOfThisLineMain (shovRulerCase: boolean; thisRulerStorage: rulerRecordPtr; thisScreenLine: lineRecordPtr): integer;
begin
HeightOfThisLineMain := 0;
if thisScreenLine <> nil then
if thisScreenLine^.lineType = rulerLineType then
begin
if showRulerCase then
HeightOfThisLineMain := thisScreenLine^.height
else
HeightOfThisLineMain := 0
```
end
else if thisScreenLine^.lineType = dummyLineType then
  HeightOfThisLineMain := thisScreenLine^.height
else if (thisScreenLine^.lineType = textPortExtendLineType) then
  HeightOfThisLineMain := thisScreenLine^.height
else
  begin
    with thisRulerStorage do
      if spaceType = singleSpace then
        HeightOfThisLineMain := thisScreenLine^.height
      else {Double spacing }
        HeightOfThisLineMain := 2 * thisScreenLine^.height
  end;
end;

B.2.6.3 Locating the Line Pointed by the Mouse in an Opened Port

The formal parameter v gets from the actual parameter the distance from the top
of the window to the top of the opened port. newLineBdaryH and leftBdaryH are
hold the distance of the indentation marker (left black arrow) and the left margin
marker (left black triangle) from the start of the ruler that contains the opened
port.

▷ function InExpandPortRulerRegionMain(□ : lineRecordPtr; [] : integer; []:
  point): boolean; «
function InExpandPortRulerRegionMainCthisScreenLine: lineRecordPtr;
  v: integer;
  mouseLoc: point): boolean;
var
  thisRulerStorage: rulerRecordPtr;
  tempScreenLine: lineRecordPtr;
  found: boolean;
  d: integer;
begin
  thisRulerStorage := FindRulerByScreenLineMain(thisScreenLine);
  InExpandPortRulerRegionMain := false;
  if mouseLoc.h >= thisRulerStorage^.newLineBdaryH then
    { move into the opened port }
  with thisScreenLine^.startBlck^.newWd do
    with textPart^ do
      begin
        found := false;
        thisRulerStorage := thisTitle^.newWd^.rulerPart;
        tempScreenLine := screenTable^.down;
        while (not found) and (tempScreenLine^.down <> nil)
          and (v < mouseLoc.v) do
          begin
            d := HeightOfThisLineMain(showRuler,
              thisRulerStorage, tempScreenLine);
          end;
      end;
if tempScreenLine^.lineType = rulerLineType then
  thisRulerStorage :=
  tempScreenLine^.startBlck^.newWd^.rulerPart;
if d = 0 then
  tempScreenLine := tempScreenLine^.down
else if (mouseLoc.v >= v) and (mouseLoc.v <= (v + d)) then
  found := true
else begin
  v := v + d;
  tempScreenLine := tempScreenLine^.down;
end;
end;

B.2.6.4 Checking Whether the Mouse is Within a Ruler

function InRulerRegionMain ([]: wdPtr; []: point): boolean;

function InRulerRegionMain (curWdStorage: wdPtr;
mouseLoc: point ): boolean;

var
  thisScreenLine: lineRecordPtr;
  v, d: integer;
  found: boolean;
  thisRulerStorage: rulerRecordPtr;
begin
  with curWdStorage^ do
    with textPart^ do
      begin
        InRulerRegionMain := false;
        if showRuler then
          begin
            thisScreenLine := topScreenLine;
            thisRulerStorage := FindRulerByScreenLineMain(thisScreenLine);
            v := topLineV;
            found := false;
            while (not found) and (thisScreenLine <> nil)
              and (v < bottomV) and (v < mouseLoc.v) do
              begin
                d := HeightOfThisLineMain(showRuler,
                  thisRulerStorage, thisScreenLine);
                if thisScreenLine^.lineType = rulerLineType then
                  thisRulerStorage :=
                  thisScreenLine^.startBlck^.newWd^.rulerPart;
                if d = 0 then
                  thisScreenLine := thisScreenLine^.down
                else if (mouseLoc.v >= v) and (mouseLoc.v <= (v + d)) then
                  found := true
              end;
          end;
      end;
end;
found := true
else begin
  v := v + d;
  thisScreenLine := thisScreenLine\.down;
end;
end;
end;
if found then
if thisScreenLine\.startBlck\.newWd <> nil then
  if thisScreenLine\.startBlck\.newWd\.theWdType = rulerWdType then {ruler}
    InRulerRegionMain := true
  else if thisScreenLine\.startBlck\.newWd\.theWdType = portWdExpandType then
    { Mouse inside opened port }
    InRulerRegionMain :=
      InExpandPortRulerRegionMain(thisScreenLine, v, mouseLoc)
end
end;

B.2.6.5 Leftmost of Indentation and Left Margin

B.2.6.6 Mapping from "a" to "b"
innerTLcornerH := tlh - leftH;
if showRuler then
  v := rulerHeight
else
  v := 0;
innerTLcornerV := tlv - v - dummyLineHeight;
end;
end;

procedure ConvertExpPortTitleToScreenCoorMain (thisTextStorage: textRecordPtr; tlh, tlv: integer; var titleTLcornerH, titleTLcornerV: integer);

var
  thisH, v: integer;
  titleRulerStorage: rulerRecordPtr;

begin
  with thisTextStorage' do
  begin
    if showRuler then
      v := rulerHeight
    else
      v := 0;
    titleRulerStorage := thisTitle^.newWd^.rulerPart;
    thisH := RulerLeftBdaryMain(titleRulerStorage);
    titleTLcornerH := tlh + titleRulerStorage^.newLineBdaryH + windowExtraH - thisH;
    titleTLcornerV := tlv - v
  end
end;

procedure ConvertExpPortCoorToScreenCoorMain (innerWindowStorage: textRecordPtr; tlh, tlv: integer; var innerTLcornerH, innerTLcornerV: integer);

var
  v: integer;

begin
  with innerWindowStorage' do
  begin
    if showRuler then
      v := rulerHeight
    else
      v := 0;
    end;

    if showRuler then
      v := rulerHeight
    else
      v := 0;
end;
v := 0;
innerTLcornerH := t1h;
innerTLcornerV := t1v - v;
end;
end;

The Boolean function GETnextEvent checks whether event belongs to the set of system events (as represented by the mask everyEvent).

B.2.8 Restore System Menu

procedure RestoreMenuBar;
begin
DISPOSEMenu(fileMenu);
DISPOSEMenu(editMenu);
DISPOSEMenu(boxMenu);
DISPOSEMenu(fontMenu);
DISPOSEMenu(styleMenu);
DISPOSEMenu(pascalMenu);
DISPOSEMenu(shapeMenu);
DISPOSEMenu(fillMenu);
DISPOSEMenu(arrangeMenu);
SETMenuBar(oldMenuBar);
DRAWMenuBar
end;

procedure RestoreMenuBar;
begin
DISPOSEMenu(fileMenu);
DISPOSEMenu(editMenu);
DISPOSEMenu(boxMenu);
DISPOSEMenu(fontMenu);
DISPOSEMenu(styleMenu);
DISPOSEMenu(pascalMenu);
DISPOSEMenu(shapeMenu);
DISPOSEMenu(fillMenu);
DISPOSEMenu(arrangeMenu);
SETMenuBar(oldMenuBar);
DRAWMenuBar
end;

procedure GetSystemEvent;
begin
INITCursor; { Displays a cursor of shape arrow }
while not GETnextEvent(everyEvent, event) do
SYSTEMTask;
end;

The Boolean function GETnextEvent checks whether event belongs to the set of system events (as represented by the mask everyEvent).
BIBLIOGRAPHY


