PSI Font Editor
Implementation Notes

by
Herve Touati (INRIA)

May, 1985

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

PSI FONT EDITOR

IMPLEMENTATION NOTES

by

Herve Touati

May 1985

ABSTRACT

The PSI (Personal Sequential Inference) machine, the first major product of Japanese FGCS (Fifth Generation Computer Systems), is intended to provide a comfortable logic program development. As its bitmap display allows it, the PSI machine can use any kind of fonts for the comfort of its users. In addition to the various styles of characters provided by the machine, the user can create his own fonts by using the PSI font editor.

A first version of the PSI font editor has been implemented in ESP (Extended Self-contained Prolog). This paper describes in detail this implementation.

The object-oriented programming features of the ESP language: objects with time dependent states, object classes and inheritance among them, have been widely used in this implementation. They provide a natural framework for the description of the program.

This paper is also intended to be an evidence of what can be done in two months and a half by a non-Japanese, nonprofessional programmer, with our software tools.

CHAPTER 1 INTRODUCTION

1.1 PRESENTATION 1
1.2 REMARKS 2

CHAPTER 2 FILE SYSTEM INTERFACE

2.1 DESIGN CONSTRAINTS 3
2.2 DESIGN COMMITMENTS 3
2.3 THE CLASS CHAR_RECORD 4
2.4 THE CLASS CHAR_RECORD WITH PATTERN 5
2.5 THE CLASS AS_FONT_FILE 8
2.6 THE CLASS GENERAL_FONT 10
2.7 THE CLASS EDITED FONT 13

CHAPTER 3 DISPLAY_INTERFACE

3.1 DESIGN CONSTRAINTS 19
3.2 DESIGN COMMITMENTS 19
3.3 THE CLASS AS FONT_DISPLAY 22
3.4 THE CLASS CHAR_MODE_DISPLAY 25
3.5 THE CLASS PATTERN MODE DISPLAY 28

CHAPTER 4 GRAPHIC INTERFACE

4.1 DESIGN CONSTRAINTS 34
4.2 DESIGN COMMITMENTS 34
4.3 THE CLASS FONT_INFERIOR WINDOW 35
4.4 THE CLASS FONT SUPERIOR WINDOW 36
4.5 THE CLASS FONT MULTIPLE_SELECT MENU 36
4.6 THE CLASS FONT MULTIPLE SELECT_MULTI COLUMN_MENU . 37
4.7 THE CLASS FONT GRAPHICS WINDOW 38
4.8 THE CLASS PATTERN_DISPLAY WINDOW 40
4.9 THE CLASS DISPLAY_SAMPLE WINDOW 42
4.10 THE CLASS PATTERN REGISTERS WINDOW 44
4.11 THE CLASS FONT_DISPLAY WINDOW 48
4.12 THE CLASS HORIZONTAL CODE DISPLAY WINDOW 53
4.13 THE CLASS VERTICAL_CODE DISPLAY WINDOW 55
4.14 THE CLASS PARAMETERS_DISPLAY_MENU 58

CHAPTER 5 EDITING_FACILITIES

5.1 DESIGN CONSTRAINTS 61
5.2 DESIGN COMMITMENTS 61
5.3 THE CLASS EDIT RECORD 62
5.4 THE CLASS AS EDIT PATTERN WINDOW 72
5.5 THE CLASS EDIT PATTERN WINDOW 81

CHAPTER 6 TOP LEVEL SYSTEM

6.1 DESIGN CONSTRAINTS 86
6.2 DESIGN COMMITMENTS 86
6.3 THE CLASS FONT_TEMPORARY WINDOW 87
6.4 THE CLASS FONT_FILE MANIPULATOR 89
6.5 THE CLASS CHAR_MODE 93
6.6 THE CLASS PATTERN MODE 95
6.7 THE CLASS FONT EXECUTIVE 100
6.8 THE CLASS FONT EDITOR 104

CHAPTER 7 PROPOSED IMPROVEMENTS

7.1 INTRODUCTION 105
7.2 ALGORITHM OPTIMIZATION 105
7.3 FUNCTIONAL IMPROVEMENTS 106
7.4 NEW FEATURES 108
7.5 ADDING METAFONT FACILITIES 108

CHAPTER 1

INTRODUCTION

1.1 PRESENTATION

This paper describes the way the PSI machine font editor has been implemented (May 1985). The implementation can be decomposed into five major parts:

1. The file system interface
2. The display interface
3. The graphic interface
4. The editing facilities
5. The top level system

Each part corresponds to one of the following chapters. In each chapter are described:

1. The design principles of the corresponding part of the font editor;
2. The role assigned to each class, and the corresponding meaning of class or instance attributes and methods;
3. The way the methods are implemented, with a description of the task assigned to each local predicate.

This paper is intended to provide a useful support for any person in charge to improve this system. Although it is possible to edit reasonably quickly any kind of fonts with this current version, a lot of things are to be improved, especially certain unwise and slow algorithms, and quite a lot of clumsy details related to the user interface.

A last chapter describes some useful improvements which could be easily implemented from the current version.

1.2 REMARKS

Throughout this paper, numbers of methods and local predicates are described. Those whose meaning is obvious from their names are given without further explanation.

Arguments of methods and local predicates are classified as follows:

1. Unified arguments: they are unified with the result of a certain computation. They thus may be instantiated or not. They are prefixed by a "^".

2. Output arguments: they are supposed not to be instantiated. Instantiate them may raise an exception or cause unexpected failures. They are prefixed by a "~".

3. Input Arguments: they must be instantiated when the method or the predicate is called. They are not prefixed by anything.

CHAPTER 2

FILE SYSTEM INTERFACE

2.1 DESIGN CONSTRAINTS

It was considered that:

1. font patterns must be easily loaded from files to a bitmap area.

2. each pattern must be accessed quickly from its code.

3. on the other hand, the modification of a font must not be cause of too much overhead: the patterns coming from an editing session must be therefore essentially in their definitive format.

4. there is no reason to assume that the user will know at the beginning what will be the maximum size of the patterns of the font he is editing. Therefore these parameters must be adjusted dynamically.

2.2 DESIGN COMMITMENTS

It was decided that the standard format of the characters must be mainly suitable to the file interface system. The other parts of the system must be adapted to this format. This standard format is an instance of the class character record with pattern. The class char record is a lower level class which must be absorbed into the previous one in a next version.

Moreover, it was decided to create two different classes of fonts to handle two different kinds of tasks: the class general font and the class edited font.

An instance of the class general font has to be considered as a fixed font. It is able to read patterns from a file, load them into a bitmap area, and give their size and position quickly on request.

An instance of the class edited font is able to read patterns from a file, and store them in a entirely modular way in the main memory.

Thus modifications are straightforward. To save back into a file, it is necessary to modify the formats. Adequate methods are provided.

The classes edited_font and general_font inherit the class as_font_file which provides basic methods for writing on and reading from files.

2.3 THE CLASS CHAR_RECORD

2.3.1 Description

Instances of this classes are simple records. They correspond to the information necessary to handle a character (see "instance attributes" below).

2.3.2 Instance Attributes

An instance of this class has 6 attributes.

1. code: the code of the character.
2. font_x: the horizontal position of the pattern of this character when loaded in a bitmap area.
3. font_y: the vertical position of the pattern of this character when loaded in a bitmap area.
4. width: the width of the character.
5. height: the height of the character.
6. bias: the bias of the character. The vertical position of the character patterns on a line is computed relatively to a virtual line called the base line. Usually, patterns are supposed to be drawn just above the base line. But some common characters, as "p" or "j", naturally go under the base line. Therefore, it is necessary to have a parameter the value of which is the distance between the base line and the bottom of the pattern: the bias is this parameter.

2.3.3 Instance Methods

```
:get_code(Instance,^Code)

:set_code(Instance,Code)

:get_size(Instance,^Width,^Height,^Bias)
```

:set_size(Instance,Width,Height,Bias)

:get_attribute(Instance,^Font_x,^Font_y,^Width,^Height,^Bias)

:get_record(Record,~String)
String is instantiated by this method to the string: double_bytes:{Code,Font_x,Font_y,Width,Height,Bias}.

2.3.4 Class Methods

:create(Class,Instance,String)
String must be of the format:
double_bytes:{Code,Font_x,Font_y,Width,Height,Bias}.

2.3.5 Local Predicates

load_into_slots(Instance,String)
String must be of the format:
double_bytes:{Code,Font_x,Font_y,Width,Height,Bias}.

load_from_slots(Instance,~String)
String is of the format:
double_bytes:{Code,Font_x,Font_y,Width,Height,Bias}.

2.4 THE CLASS CHAR_RECORD_WITH_PATTERN

2.4.1 Description

This class inherits the previous one: the char_record class. One attribute has been added for storing a pattern, which is represented as a double_byte string. Two other attributes have been added, which store the position of the pattern. This appeared to be useful when the editing process make several objects exchanging instances of this class as messages.

2.4.2 Instance Attributes

An instance of this class has 3 more attributes than an instance of the inherited class char_record:

1. char_x: the horizontal position of the pattern when transmitted to or received from the object in charge of editing patterns, which is an instance of the class edit_pattern_window.

2. char_y: the vertical position of the pattern paired with the previous attribute : char_x.

3. bit_pattern: the pattern.

2.4.3 Instance Methods

:get_char_position(Instance,^Char_x,^Char_y)

:set_char_position(Instance,Char_x,Char_y)

:get_pattern_position(Instance,^Font_x,^Font_y)

:set_pattern_position(Instance,Font_x,Font_y)

:set_pattern(Instance,Pattern)

:get_pattern(Instance,~Pattern)

:format_pattern(Instance,Raster_width)

Before explaining what this method is doing, some details about the way patterns are stored in strings must be explained.

A pattern defines a rectangular area in the bitmap memory, and gives the values of each of the dots (0 or 1). It is easy to store a pattern of width Width and height Height in a single bits string: the first line of the pattern is stored from the position 0 to the position Width - 1, the second line from the position Width to the position 2 * Width - 1, and so on.

But it has been decided to use double bytes strings instead, mainly because the window graphics methods accept patterns only if stored in double bytes strings.

There are several ways to store a pattern into a double bytes string. The way which was chosen is adapted to the window graphics method formats. It consist to increase the width of the pattern by adding white columns to its left, so that the final width is a multiple of 16, or, in other words of the form: 16 * Integer. It is then stored into a double bytes string exactly the same way it would have been stored into a single bits string. The length of the double bytes string used is therefore equal to Height * Integer. The integer Integer is called the raster width of the pattern string.

The raster width can be any integer big enough to satisfy the constraint: 16 * (raster width) >= Width.

In a font, all the pattern strings have the same raster width; this uniformity reduces the access time to the patterns once stored in the bitmap memory. And it does not waste too much space, for usual fonts at least, where the size of the patterns does not vary considerably from one character to another.

When new patterns have been added, or old patterns

suppressed, the maximum width of the patterns in the currently edited font is recomputed, and the new global raster width computed from this maximum width.

It is then necessary to adapt all the records to this new raster width. It is what this method enables to do.

2.4.4 Class Methods

:create(Class,Instance)

Identical to :new(Class,Object).

2.4.5 Local Predicates

format_pattern(Instance,New_raster_width)

This is the local predicate directly called by the method of the same name. It computes the minimum possible raster width from the current width of the pattern, and pops out an error message if the value of New_raster_width is too small.

It computes the current raster width, by dividing the length of the pattern string by the total height (stored respectively in the attributes "bit_pattern" and "total_height"), and compares it to New_raster_width. Depending on the result of this comparison, it then calls the local predicate inflate, the local predicate deflate, or simply do nothing. When the predicates deflate or inflate are called, a new string is created, filled by the action of these predicates, and then becomes the new value of the attribute "bit_pattern".

deflate(Old_string,New_string,Old_width,New_width,Height,N)

Takes, in Old_string, the part corresponding to the line of height H, suppresses one or several double_bytes on the left, and stores the resulting substring into New_string. Then calls itself recursively, after having increased the height N by one. Stops when N is greater or equal to Height, the height of the pattern.

inflate(Old_string,New_string,Old_width,New_width,Height,N)

Works as the previous predicate, except that it adds white double bytes (equal to 0) at the left of each line, instead of suppressing some.

min_raster_width(Width,Min_raster_width)

error_message(Instance)

2.5 THE CLASS AS_FONT_FILE

2.5.1 Description

This class provides the basic interface functions between fonts and files. Currently, the directory accessed by the system is fixed. But, since the name of this directory appears only as the initial value of an attribute, it can be easily modified.

A font is stored on two binary files. The unit used for recording in these files is the double_byte. Here is a description of the contents of these two files:

1. Font_namehd.bin: This file, the head file, is composed of a head and a body. The head, of length 5 (5 double_bytes), contains the basic information about the size of the font: the line height, the base, the number of characters recorded, the raster width, the total height. Each of these corresponds to an instance attribute of this class. The body, of length (6 * number_of_characters), contains, for each character of the font that has been defined, the contents of a char_record: code, font_x, font_y, width, height, and bias.

2. Font_namepat.bin: This file, the pattern file, contains all the character patterns defined in the current font. Once the contents of this file are loaded into a double_bytes string, they can be directly stored into the bitmap memory without any modification. Patterns are stored one after the other. Their position and size is recorded in the corresponding char_record, in the head file.

2.5.2 Instance Attributes

An instance of this class has 11 attributes. There are: fixed strings used to compute file path names, fixed parameters giving the size of certain records, and global parameters depending on the font instance, as its name, or its size.

1. font_name: the name of the font. This is not recorded directly in the file, but, instead, appear in the names of the files.

2. directory_path: this slot has currently the fixed value: ">sys>user>font".

3. head_path: this slot has currently the fixed value: "hd". This string is appended at the end of the font name to form the name of the head file.

4. pattern_path: this slot has currently the fixed value: "pat". This string is appended at the end of the font name to form the name of the pattern file.

5. file_head_size: this slot has also a fixed value : 5. This is the length of the head part of the head file. If this has to be modified, the local predicates set_head, get_head must be also modified.

6. char_record_size: this slot also has a fixed value: 6. This is the length of a string corresponding to the contents of a char_record instance.

7. line_height: this slot is a parameter of a font. It is the distance between two lines of characters, when writing with this font.

8. base: this slot is also a parameter of a font. It corresponds to the distance between the top of a line of characters and the base line, virtual line that is used for precise alignment of the characters in a line.

9. number_of_char: the number of characters defined in the font.

10. raster_width: the global raster width of the font. Look at the class char_record_with_pattern for more details about it.

11. total_height: This is also a global parameter of a font. In a font, the number of double bytes used to store a line of pattern is the same for all the patterns, and is equal to raster_width. When the contents of the pattern file are loaded into the bitmap memory, the logical space they use is a rectangle of width 16 * raster_width, and of height total_height. Total_height is also, of course, the sum of the heights of all the patterns recorded in the font.

2.5.3 Instance Methods

:load_from_file(Instance,~Char_string,~Pattern_string)
Loads the head of the head file into the 5 corresponding slots; the rest of the head file (the character records) are left one after the other and put in the string Char_string. The contents of the pattern file are put in the string Pattern_string.

:save_into_file(Instance,Char_string,Pattern_string)
Does exactly the opposite of what load_from_file does. Creates new files each time, to avoid possibly dangerous destructive actions.

:set_head(Instance,Head_string)
Head_string must be a double_bytes string of length 5. The contents of this string become the values of the following slots in this order: line_height, base, number_of_char, raster_width, total_height.

:get_head(Instance,~Head_string)

Creates a new string, a double_bytes string of length 5, and stores in it the contents of the five slots mentioned above, in that order.

2.5.4 Local Predicates

set_head(Instance,Headstring)
(See the corresponding instance method)

get_head(Instance,~Headstring)
(See the corresponding instance method)

load_head(Instance,Path_name,~Char_string,^Pattern_file_size)
Given the path name of the head file, reads it, puts its head into slots using set_head, puts its body into a new string: Char_string, and computes the size of the pattern file, which is merely: total_height * raster_width.

save_head(Instance,Path_name,Char_string)
Does exactly the opposite of what load_head does. Uses get_head instead of set_head.

load_pattern(Instance,Path_name,~Pattern_string)
Given the path name of the pattern file, reads it, and puts its contents into a new string: Pattern_string.

save_pattern(Instance,Path_name,Pattern_string)
Does exactly the opposite of what load_pattern does.

path_head(Instance,~Head_path)
Computes the head file path from the values of the instance slots, by appending strings.

path_pattern(Instance,~Pattern_path)
Computes the pattern file path from the values of the instance slots, by appending strings.

path(Directory,Font_name,Postfix,~Path_name)
Simply appends the strings Directory, Font_name, Postfix and ".bin", and puts the result into Path_name.

2.6 THE CLASS GENERAL_FONT

2.6.1 Description

This class inherits two classes: the class as_font_file, described above, and the class as_general_font, which is provided by the window system.

The source code of the class as_general_font is less than one page long, and has been very clearly written and documented by Iima san.

The general_font class itself is a very simple one. At creation of an instance, a font name must be specified. During the initialization, the two font files corresponding to the given name are read; character records are stored in a hash_index, using the codes as keys, while the patterns are kept in one large string, saved as an attribute. This string can be stored into the bitmap memory as it is.

2.6.2 Instance Attributes

An instance of this class has, outside the inherited ones, 3 attributes.

1. char table: this slot is a hash_index which contains the necessary information about each existing character of the font. For each character, a char_record instance is created, filled with the information contained in the head file on this character, and then stored into this hash_index. The key used is the code of the character.

2. pattern string: this string contains the contents of the pattern file, without any modification.

3. code list: this slot is an instance of the class list, and contains the list of the codes of the existing characters.

2.6.3 Instance Methods

:get_code_list(Instance,^Code_list,^Number_of_char)
This, and all the following methods, unless otherwise specified, are trivial methods, used only for direct information retrieval. Their meaning is apparent from their names.

:get_code_list(Instance,^Code_list)

:get_size(Instance,Code,^Width,^Height,^Bias)
When the code is the code of the space character, and when the space character does not exist in the font, default values for the width, the height and the bias are computed from the global parameters of the font. When setting a new font to a window, the size of the cursor is modified and computed from

the size of the space character. Without these default values, it would not be possible to use a font that does not have a space character.

:get_attribute(Instance,Code,^Font_x,^Font_y,
^Width,^Height,^Bias)

:get_attribute(Instance,Code,^Font_bitmap_area,
^Font_x,^Font_y,^Width,^Height,^Bias)
Font_bitmap_area is a bitmap area kept as an instance attribute of the inherited class as_general_font. When not used as an area, this attribute has the value 'nil'. When used as an area, the area contains all the patterns of the existing characters. Font_x and Font_y are the coordinates of the top left corner of the character pattern corresponding to the code Code in the bitmap memory. Width and Height give the size of this pattern.

:load_font(Instance)
creates a bitmap area, loads the contents of the string "pattern_string" in it, in other words, loads all the existing character patterns in it, and keeps this area as an instance attribute (an inherited one, from the class as_general_font).

:standard_line_and_base(Instance,^Line_height,^Base)

:in(Instance,Code)
Checks if the code Code corresponds to a character in the font Instance. If not, fails. Otherwise, succeeds.

:initialize(Instance)
reads from file the contents of the font. The head of the head file is put into slots (this is done using inherited as_font_file methods and slots). The records contained in the body of the head file are distributed among char_record instances, which are in turn saved in the hash_index "char_table". The contents of the pattern file are just loaded into a string, which becomes the value of the attribute "pattern_string".

:get_size(Instance,Code,^Width,^Height)

:get_width(Instance,Code,^Width)

:get_height(Instance,Code,^Height)

:load(Instance)
Identical to :load_font.

:get_character(Instance,Code,^Font_x,^Font_y,
^Width,^Height,^Bias)

:get_character(Instance,Code,^Font_bitmap_area,
^Font_x,^Font_y,^Width,^Height,^Bias)
Identical to :get_attribute.

:character_line(Instance,^Line_height,^Base)

2.6.4 Class Method

:create(Class,^Instance,Font_name)
Font_name must be instantiated to a string, corresponding to a currently available font. This method creates an instance of this class, sets its name to be Font_name, and then calls the instance method :initialize.

2.6.5 Local Predicates

load_char_records(Instance,Char_string)
Char_string must contain the contents of the body of the head file. In that case, it is composed of a series of 6 double_bytes records, in a format adapted to char_record instances. This predicate creates for each record a char_record instance, in which the contents of the record are stored. It creates also a hash_index, in which it stores each record. The key used is the code of the record. It creates also a list, in which it saves the code of each of the records. When all the records from the string Char_string have been saved that way, the resulting hash_index and list become the values of the attributes "char_table" and "code_list" respectively.

load_char_records(Instance,Table,List,Char_string, Length,N)
Takes the part of the string Char_string located from N to N + 6, saves it in a new char_record instance, puts this instance into the table Table, puts the corresponding code into the list List, and calls itself recursively after having increased N by 6. Stops when N is greater than or equal to Length, the length of the string Char_string.

almost_prime(Integer,^Prime)
Computes the first prime number equal to or greater than Integer which can not be divided by 2, 3, 5, and 7. It is used for adjusting the size of an hash_index.

2.7 THE CLASS EDITED_FONT

2.7.1 Description

As the class general_font, this class inherits the two classes as_font_file and as_general_font.

The main difference between the class edited_font and the class general_font is that the former must allow easy modifications of the set of characters, while the latter is fixed, and must allow quick access to the characters.

To allow quick access to the characters, an instance of the class general_font keeps records of the parameters of each character in char_record instances stored in a hash_index, while the character patterns are loaded together into the bitmap memory. Among the parameters recorded in char_record instances, two specify the absolute position of the character pattern in the bitmap memory.

On the other hand, to allow easy modifications, it is better to save each character independently. This is what is done in an instance of the class edited_font. However, this modularity in the way of storing characters has cannot be preserved when it is to save the current state of the edited_font instance into a file: the absolute positions that the character patterns will have, when stored later into the bitmap memory, must be computed, as well as the font global parameters like the line_height or the raster_width.

2.7.2 Instance Attributes

An instance of this class has, outside the inherited ones, 3 attributes.

1. char_table: this slot is a hash_index which contains the necessary information about each existing character of the font. For each character, a char_record_with_pattern instance is created, filled, on one hand with the information contained in the head file on this character, and, on the other hand, with the corresponding pattern read from the pattern file. This record is then saved into this hash_index. The key used is the code of the character. Note that, at the difference of general_font instances, edited_font instances store every character as a whole in a record, and do not put apart the pattern.

2. code_list: this slot is an instance of the class list, and contains the list of the codes of the existing characters.

3. espacement: this attribute is a fixed one, and is used for computing the line_height from the maximum height of the existing characters. It can be either an integer E or a list of two integers [N,E]. E alone means that N is defaulted to 1. The formula used is:
line_height = (Max_height * N) + (Max_height / E) + 1.

2.7.3 Instance Methods

:get_name(Instance,^Font_name)

:in(Instance,^Code)

:load(Instance)

reads from file the contents of the font. The head of the head file is put into slots (this is done using inherited as_font_file methods and slots). Each record contained in the body of the head file contains the address, in the pattern file, of the associated pattern.

The contents of the pattern file once put into a string, the pattern corresponding to a record begins at the position raster_width * font_y and is of length raster_width * height, where raster_width is a global parameter of the font, already read from the head of the head file, and font_y and height are parameters contained in each record.

For each record, a new instance of the class char_record_with_pattern is created, in which the record itself as well as the corresponding pattern are saved. These char_record_with_pattern instances are in turn saved in the hash_index "char_table". Moreover, the codes of all the existing characters are stored into the list "code_list".

:set_record(Instance,Record,Code)

Record must be a character_record_with_pattern instance or the atom 'nil'. This method makes a copy of the record Record, adds this copy to the hash_index "char_table" at the key Code, erasing the previous record if any, and updates the list "code_list". In the case Record is equal to 'nil', however, set_record has the same effect as the method :delete_record. It is necessary to make a copy of the record, because the record Record may be modified later by another object, and the user may obtain that way a pattern different from the one he wanted to save.

:delete_record(Instance,Code)

Removes the record stored in the table "char_table" at the key Code, if any, and removes the code Code from the list "code_list" if necessary.

:get_record(Instance,~Record,Code)

If no record is found at the key Code, this method, instead of failing, instantiates Record to the atom 'nil'.

:get_pattern(Instance,Code,~Pattern_string)

This raises an error and stops the program when Code does not correspond to an existing character. It can be thought as a logical bug and must be fixed, even if it has never been used in that case until now.

:raster_width_and_max_height(Instance,^Raster_width, ^Max_height)

Computes the maximum width of the existing characters, and the highest and the lowest positions the characters will take on a same line. Instantiates Max_height to the the difference between the highest and the lowest positions, and Raster_width to the smallest integer satisfying the inequality:

16 * Raster_width >= maximum width.

:save(Instance)

Computes the global parameters of the font: the line height,

the base, the number of characters, the raster width and the total height. Then adjusts the raster width of all the characters to be equal to the global raster width. It means that all the patterns will use exactly the same memory space for saving a line. Then computes the absolute positions the patterns will have when loaded in the bitmap memory, and puts this information into the records. Then transfers the contents of these records into two strings, storing the patterns separately, and finally stores these two strings into two different files, namely the head and the pattern files.

2.7.4 Classes Method

:create(Class,~Instance,Font_name)

Font_name must be instantiated to a string, corresponding to a currently available font. This method creates an instance of this class, and sets its name to be Font_name.

2.7.5 Local Predicates

load_into_slots(Instance,Char_string,Pattern_string)

Char_string must contain the contents of the body of the head file, and Pattern_string the contents of the pattern file. In that case, Char_string is composed of a series of 6 double_bytes records, in a format adapted to char_record instances. This predicate creates for each record a char_record_with_pattern instance, in which the contents of the record, as well as the associated pattern, are stored. It creates also a hash_index, in which it stores each record, with key the code of the record. It creates also a list, in which it saves the code of each of the records. When all the records from the string Char_string have been saved that way, the resulting hash_index and list are set as attributes, respectively the "char_table" and "code_list" attributes.

put_into_slots(Instance,Char_string,Pattern_string,
Table,List,N)

Takes the part of the string Char_string located from N * 6 to (N + 1) * 6, saves it in a new char_record_with_pattern instance, and takes, from this new record, the parameters font_y and height. Then takes the pattern corresponding to this record from the Pattern_string. This pattern is located from the position font_y * raster_width, and its length has for value height * raster_width, where raster_width is a global parameter of the font. Loads this pattern into the record, adds the record in the table Table, puts the corresponding code into the list List, and calls itself recursively after having increased N by 1. Stops when N is greater than or equal to the global parameter "number_of_char".

set_global_parameters(Instance)

computes the 5 global parameters of the Font, and stores them in the following slots: "line_height", "base", "number_of_char", "raster_width" and "total_height".

maximum_width_and_height(Instance,^Raster_width,^Max_height)
computes the maximum width of the characters, as well as the highest and the lowest positions the characters will take on a same line. Instantiates Max_height to the difference between the highest and the lowest positions. Max_height is not generally equal to the maximum height of the character patterns, but it is the relevant parameter to use for computing the line height. Raster_width is the smallest integer such that:
16 * Raster_width >= maximum width.

compute_parameters(Table,List_of_codes,Highest_point,
Lowest_point,Max_width,Total_height,N)
Removes the next code in the list List_of_codes, computes the highest and lowest points of the character of code Code, as well as its width. Modifies the values of Highest_point, Lowest_point and Max_width is necessary, adds to Total_height the height of the character, and calls itself recursively. Stops when List_of_codes becomes empty.

format_bit_patterns(Instance)
modifies the raster width of the character patterns so that it be equal to the global raster width.

format_patterns(Table,List_of_codes,Raster_width)
removes the next code from the list List_of_codes, gets from the table Table the character of code Code, modifies the raster width of the pattern of this character, and calls itself recursively. Stops when the list List_of_codes becomes empty.

load_from_slots(Instance,~Char_string,~Pattern_string)
Loads all the existing characters into two strings: Char_string and Pattern_string. The formats correspond for Char_string to the format of the body of the font head file, for Pattern_string to the format of the font pattern file.

take_from_slots(Instance,Char_string,Pattern_string, List_of_codes,
Char_string_pointer,Pattern_string_pointer)
Removes the next code from the list List_of_codes, and takes from the table "char_table" the corresponding record. Puts the head of this record in a string, and puts this string into Char_string from the position Char_string_pointer; takes the pattern of this record and puts it into Pattern_string from the position Pattern_string_pointer; adds 6 to Char_string_pointer, the height of the record to Pattern_string_pointer, and calls itself recursively. Stops when the list becomes empty.

almost_prime(Integer,Prime)
The same as the almost_prime local predicate of the class general_font.

espacement(Max_height,Espacement,Line_height)
See the instance attribute espacement of this class.

raster_width(Width,Raster_width)
The same as the min_raster_width local predicate of the class char_record_with_pattern.

copy_record(Record,Copy)

CHAPTER 3

DISPLAY_INTERFACE

3.1 DESIGN CONSTRAINTS

It has been decided that:

1. The font editor must have two modes of display, namely the character mode display and the pattern mode display.

2. The character mode display is in charge to display all the characters of a font. In Japanese standards, a font may have up to 65536 characters. Therefore, the character mode display cannot display all the characters of a font at the same time on the screen. It must provide scrolling facilities allowing the user to choose which characters have to be displayed on the screen. Moreover, the character mode display must also provide facilities to create new characters by assigning a pattern to a code, to reuse old patterns, to load a font from a file and to save a font into a file. What the commands are doing must be as obvious as possible. The final design objective of the system must be to make the user feel he is editing a font, manipulating directly patterns, codes, files, fonts and characters, not using a cumbersome program.

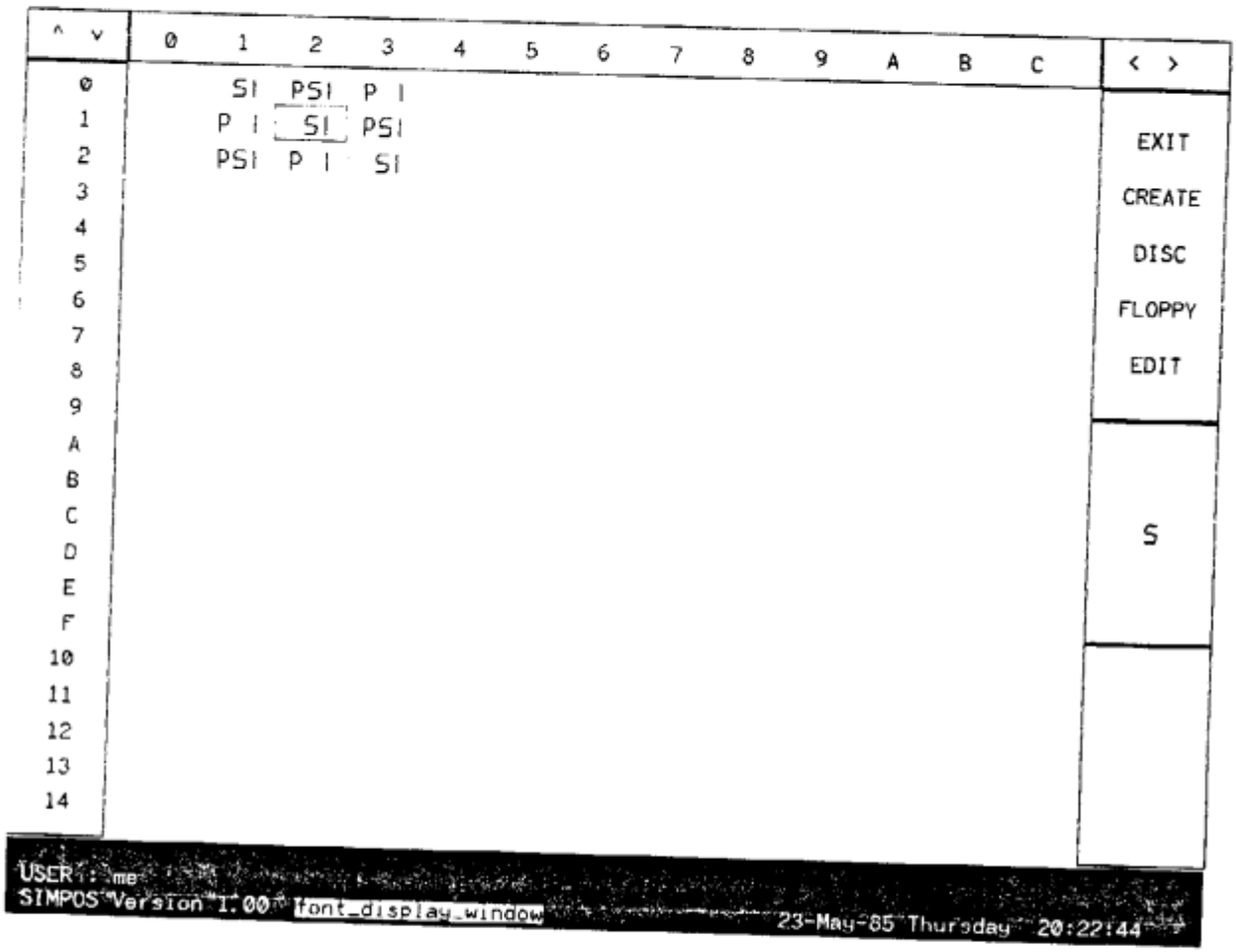
3. The pattern mode display is in charge to display a pattern in such a way it can be easily edited dot by dot. This implies that each dot of the pattern must be displayed as a square of several dots wide. Moreover, it must be possible to display this pattern at any time in normal size, and to cut and reuse parts of it easily. It means that registers for storing and displaying parts of characters should be made available.

3.2 DESIGN COMMITMENTS

The character mode display and the pattern mode display both use a full screen.

The character mode display is composed of: two menus to enter the scrolling orders; one menu for other orders such as the change mode order, or the access to file orders; two windows for communicating with

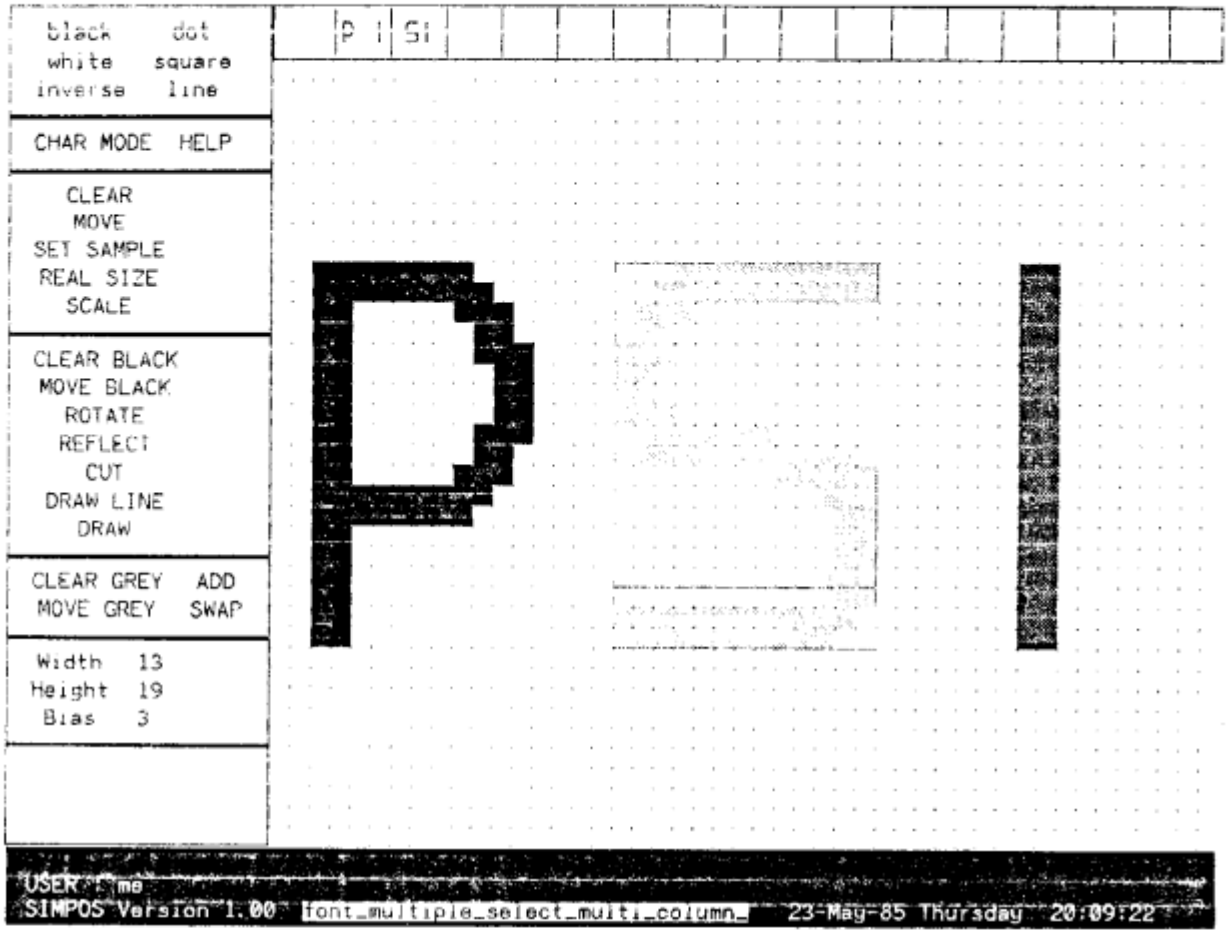
the pattern mode display; two windows for displaying the 8 lower and the 8 upper bits of the character codes; and, one window for displaying characters. Their relative position can be examined on the following hard copy of the PSI screen.



Menus are created first. Their size is automatically computed by the window system. Once having decided that the two windows in charge of assuring the communication with the pattern mode, should have the same size, the sizes of the menus completely determine the sizes and positions of all the windows and menus of this mode.

The pattern mode display is composed of: 5 menus, mainly for editing functions; one window for displaying some parameters of the pattern being edited; one window for displaying the pattern being edited in real size, or a line of characters picked up from the current font, also in real size; one window for displaying the contents of pattern registers; and one window for displaying the currently edited pattern, showing its dots as squares of several dots wide (from 5 to 25 dots, at will).

The pattern mode display is a little more complex than the character mode display. The reason is that the window used for editing patterns must have its width and its height equal to a multiple of the size of the squares used for representing dots of patterns. The size of



these squares is an important parameter, called the grid size. The common width of the menus is the smallest integer acceptable as a width for all the menus such that the width of the window used for editing patterns is a multiple of the grid size. The height of the window with registers for storing patterns is the smallest integer such that each register is allocated enough place on the window screen for displaying the contents of the screen of the window editing patterns, and such the height of the editing window is a multiple of the grid size.

When the grid size has to be modified at the user's request, menus are killed, recreated, and the sizes and positions of all the windows and menus are recomputed with the new grid size.

The class as_font_display is in charge to provide, to the classes which inherit it, basic iterative instance methods for handling menus and windows.

The class char_mode_display inherits the class as_font_display; an instance of this class is in charge of creating the windows and the menus of the character mode display, and of setting their sizes and positions.

The class pattern_mode_display inherits the class as_font_display; an instance of this class is in charge of creating the windows and the

menus of the pattern mode display, and of setting their sizes and positions.

3.3 THE CLASS AS_FONT_DISPLAY

3.3.1 Description

This class is a really simple one. It gathers attributes common to the character mode display and the pattern mode display, such as the size of the screen, or a global superior window. It provides basic iterative methods for handling all the menus and all the windows of a display mode at the same time.

3.3.2 Instance Attributes

An instance of this class has 4 attributes.

1. screen width: a fixed attribute, initialized at the creation of an instance.

2. screen height: also a fixed attribute, initialized at the creation of an instance. It must be noted that the screen height is not the same for a Mitsubishi PSI and an Oki PSI.

3. font: this attribute has for default value the current default font: font_13. This attribute may not be used anywhere. This point must be checked.

4. superior: is set to be a superior window when an instance of this class is created. This superior window is used by the top level loop for reading inputs from every window or menu without having to know from which of them the input is coming. It must be declared as superior window of all the windows and menus of the two display modes. This is done by using a method of this class.

3.3.3 Instance Methods

:initialize(Instance)
Creates all the menus and windows of any of the two display modes, and set their sizes and positions. The menus are treated first, and some global parameters are saved as attributes of the display modes, before being used for computing the sizes and the positions of the windows.

:show(Instance)
shows all the windows and menus of a mode display.

:draw(Instance)
clears and redraws the display of all the windows of a mode display.

:reshape(Instance)
Is only used by the pattern mode display when the grid size has to be changed. Kills the menus, recreates them, adjusts their widths to the new grid size, and adjusts the sizes and positions of the windows to the current grid size. This is not optimized: menus are killed and recreated only to compute their minimum sizes. It will be better to save at first their minimum sizes in slots, and to modify only their widths when the grid size is changed.

:kill(Instance)
kills all the windows and menus of a mode display.

3.3.4 Class Method

:create(Class,~Instance,Superior_window)
creates a new instance of this class, and initializes the slots. The superior window Superior_window is saved in the slot "superior".

3.3.5 Local Predicates

initialize_menus(Instance)
Asks to the mode display the list of the names of the menus; creates these menus, asks to the mode display to set their sizes and positions, and loads their names in their name slots. The names are used in the top level loop to determine from which window or menu the last input is coming.

initialize_windows(Instance)
Asks to the mode display the list of the names of the windows; asks to the mode display what must be their sizes and positions; creates these windows, and loads their names in their name slots. The names are used in the top level loop to determine from which window or menu the last input is coming.

create_menus(Instance,List_of_menus_names)
Removes from the list List_of_menus_names the next menu name, asks to the display mode what are the commands proposed by this menu, creates it, with the contents of the slot "superior" as superior window, saves it in a slot of the mode display, and calls itself recursively. Stops when the list of names is empty.

create_windows(Instance,List_of_windows_names)
Removes from the list List_of_windows_names the next window name, asks to the display mode the size and position of this window, as well as its nature, creates it, with the contents

of the slot "superior" as superior window, saves it in a slot of the mode display, and calls itself recursively. Stops when the list of names is empty.

give_names(Instance,List_of_names)
Removes from the list List_of_names the next menu or window name, asks to the display mode the menu or window of that name, puts that name in a slot of that menu or window, and calls itself recursively. Stops when the list of names is empty.

show(Instance,List_of_names)
Removes from the list List_of_names the next menu or window name, asks to the display mode the menu or window of that name, shows that menu or window, and calls itself recursively. Stops when the list of names is empty.

draw(Instance,List_of_windows_names)
Removes from the list List_of_windows_names the next window name, asks to the display mode the window of that name, refreshes and redraw the display of that window, and calls itself recursively. Stops when the list of names is empty.

kill(Instance,List_of_names)
Removes from the list List_of_names the next menu or window name, asks to the display mode the menu or window of that name, kills that menu or window, and calls itself recursively. Stops when the list of names is empty.

redraw_menus(Instance)
Asks to the mode display the list of the names of the menus; kills these menus, and calls the local predicate initialize_menus.

reshape_windows(Instance)
Asks to the mode display the list of the names of the windows; asks to the mode display what must be their sizes and positions, adjusted to the value of a new grid size; modifies the sizes and positions of these windows following these adjustments.

reshape_windows(Instance,List_of_windows_names)
Removes from the list List_of_windows_names the next window name, asks to the display mode the window of this name, as well as its new size and position; momentarily puts this window at the position (0,0), changes its size to its new value, and then its position to its new value; then calls itself recursively. Stops when the list of names is empty. To set the window momentarily to the position (0,0) is necessary because the new size may be larger than the old one, and the window manager does not allow a window to go out of the screen. To modify the position first solves the problem for windows having their sizes increased, but creates an identical problem for windows having their sizes decreased.

3.4 THE CLASS CHAR_MODE_DISPLAY

3.4.1 Description

This class inherits the previous one: the as_font_display class. 12 attributes have been added, 3 corresponding to menus, 5 to windows, and 4 to margins used to compute the sizes and the positions of the windows and menus. An instance of this class keeps all the information concerning the contents of the character mode menus, as well as the sizes and positions of the menus and windows of this mode. Otherwise, the main responsibility of this class is to maintain the coherence of the display of the characters. Characters are composed of a pattern and a code; the pattern, the 8 lower bits and the 8 higher bits of the code are displayed separately, in three different windows. An instance of this class assures that patterns are correctly displayed in rows and columns, that to each row and to each column corresponds a 8 bit number, which is displayed at one extremity of it, and that the 8 bit number corresponding to the row in which a pattern of a character is displayed coincides with the higher 8 bit part of its code, and that the 8 bit number corresponding to the column in which a pattern of a character is displayed coincides with the lower 8 bit part of its code.

3.4.2 Instance Attributes

1. horz_scroll_menu: small menu at the top right of the display (see figure above) proposing two horizontal scrolling orders: "<" or scrolling to see one page of display hidden on the left, and ">" or scrolling to see one page hidden on the right.

2. vert_scroll_menu: small menu at the top left of the display (see figure above) proposing two vertical scrolling orders: "^" or scrolling to see one page of display hidden at the top, and "v" or scrolling to see one page hidden at the bottom.

3. command_menu: menu offering the major commands of the character mode: "EXIT" for quitting the font editor, "DISC" for the interface with disc files, "FLOPPY" for communicating with floppy files (not yet implemented) "EDIT" for selecting the pattern mode.

4. horz_code_window: horizontal window at the top of the screen which displays the lower 8 bits of the character codes being displayed on the screen. This window is only affected by the horizontal scrolls.

5. vert_code_window: vertical window on the left of the screen which displays the higher 8 bits of the character codes being displayed on the screen. This window is only affected by the vertical scrolls.

6. font_display_window: biggest window of the screen, in charge of displaying the patterns of the characters of a font.

7. grey_window: window in charge of the communication with the pattern_mode. In the pattern_mode, the window used for editing patterns can display two patterns at the same time: one in grey, the other in black. The pattern displayed in this window will be displayed in grey in the editing window when the user selects the pattern mode; inversely, the pattern displayed in grey in the pattern mode will be displayed in this window when the user comes back to the character mode.

8. black_window: window in charge of the communication with the pattern_mode. In the pattern_mode, the window used for editing patterns can display two patterns at the same time: one in grey, the other in black. The pattern displayed in this window will be displayed in black in the editing window when the user selects the pattern mode; inversely, the pattern displayed in black in the pattern mode will be displayed in this window when the user comes back to the character mode.

9. left_margin:is set to the width of the menu "vert_scroll_menu", and then used for computing the sizes and positions of the windows and the positions of the other menus.

10. right_margin:is set to the width of the menu "horz_scroll_menu", and then used for computing the sizes and positions of the windows and the positions of the other menus. It must be noted that its value is the width of the menu "horz_scroll_menu", not the distance separating this menu and the left edge of the screen.

11. upper_margin:is set to the maximum of the heights of the menus "vert_scroll_menu" and "horz_scroll_menu", and used for computing the sizes and positions of the windows and the position of the other menu.

12. lower_margin:is set to the sum of the heights of the menus "vert_scroll_menu" and "horz_scroll_menu", and used for computing the sizes and positions of the windows "grey_window" and "black_window".

3.4.3 Instance Methods

:menus_list(Instance,^List_of_menu_names)

The names used for the menus are the atoms used for referencing them as instance attributes of this class. This is quite convenient, since ESP allows to call an instance attribute using a variable instantiated to the name of the attribute, as in the following example:
X = black_window, Black_window = Instance!X.

:windows_list(Instance,^List_of_window_names)

The names used for the windows are their names as attributes. The reason is the same as in the case of the menus.

:items_list(Object,Menu_name,^Items)
Gives the items list and the class name of the menu of name Menu_name in the following format:
Items = (Class_name,Items_list).
The items list is in the format required by the window system for menus.

:parameters_list(Instance,Window_name,^Parameters)
Gives the class name, the size and the position of the window of name Window_name, in the following format:
Parameters = (Class_name,[position(X,Y),size(Width,Height)]).

:refresh(Instance)
Recomputes the maximum width and height of the character patterns of the currently edited font; then recomputes the minimum distance between two consecutive patterns on the same line and on the same row, in the window "font_display_window". Then asks to the windows "horz_code_window" and "vert_code_window" the present distance between two consecutive codes. With these data, computes an acceptable distance between two consecutive rows and two consecutive columns, and gives this information to the three windows which need it: "font_display_window", "horz_code_window" and "vert_code_window". Then clears the displays of these three windows, and redraws them with the newly computed distances between the rows and between the columns.
Everything is ready to ensure that the distance between two consecutive rows or columns will always be the minimum one. But there is a logical bug in this part of the program: for the distance between two consecutive rows or two consecutive columns of character patterns to be always the minimum acceptable one, the methods :get_espacement of the window classes horizontal_code_display_window and vertical_code_display_window, should be modified in such a way they would always give the minimum distance between two consecutive codes, and no the distance currently used, as they do in this version.

before:draw(Instance)
The :draw method is inherited from the class as_font_display. This before demon asks to the windows "horz_code_window" and "vert_code_window" the current distance between two consecutive codes, to the window "font_display_window" the minimum distance between two consecutive patterns on the same row or on the same column; computes from these data an admissible distance between two rows and between two columns, and adjust the internal parameters of these three windows to these values.

3.4.4 Class Method

after:create(Class,~Instance,Superior_window)
The :create method is inherited from the class as_font_display. The after demon calls another method inherited from this class, the instance method :initialize.

3.4.5 Local Predicates

set_margins(Instance)
This method must be used just after the menus are created. Computes the four margins "left_margin", "right_margin" "upper_margin" and "lower_margin" from the minimum sizes the window system gave to the menus when they are created. Using these margins, sets the sizes and the positions of the menus.

set_espacement(Instance)
Is the local predicate called by the before demon before:draw.

menus_list(List_of_menus_names)

windows_list(List_of_windows_names)

items_list(Menu_name,(Class_name,Items_list))

parameters_list(Instance,Window_name,(Class_name,
[position(X,Y),size(Width,Height)]))
Computes the size and the position of the window of name Window_name using only the screen size, and the four margins saved as attributes. Must be therefore executed after the local predicate set_margins. The computations are essentially trivial ones.

3.5 THE CLASS PATTERN_MODE_DISPLAY

3.5.1 Description

This class inherits the class as_font_display. 14 attributes have been added, 6 corresponding to menus, 3 to windows, 3 to margins used to compute the size and the positions of the windows and menus. Another one corresponds to the grid size, which is the size of the squares used to represent one dot of a pattern being edited, and another to the number of pattern registers displayed at the top of the screen in pattern mode.

It is very similar to the class char_mode_display in the way of creating and setting the sizes and positions of menus and windows. One important difference however, comes from the constraint that the window used for editing patterns must have its width and height equal to a multiple of the grid size.

Moreover, since the user is supposed to be allowed to change the

grid size at will, a method must be provided for that.

3.5.2 Instance Attributes

1. brush menu: menu at the top left of the display (see figure above) proposing three different "colors": "BLACK", "WHITE" and "INVERSE", and three different "shapes": "DOT", "SQUARE" and "LINE" for the brush. The brush is nothing else than the mouse the user uses like a brush when drawing patterns in the "pattern_window" (see below).

2. outside menu: menu just under the previous one, proposing either to come back to the character mode by the command "CHAR MODE" or to give some help to the user by the command "HELP". The help currently available is only a moral help.

3. global menu: menu lying just under the previous one, and proposing various commands to modify the current display: "CLEAR" clears the display of the window "pattern_window"; "MOVE" translates the display of the window "pattern_window"; "SET SAMPLE" writes in the window "sample_window" in real size a line of several characters of the current font; "REAL SIZE" writes in real size in the window "sample_window" one of the two patterns contained in the window "pattern_window"; and "SCALE" changes the grid size.

4. black menu: menu lying just under the previous one, and proposing various commands to modify the pattern displayed in black in the window "pattern_window": "CLEAR BLACK" kills this pattern; "MOVE BLACK" translates this pattern; "CUT" cuts a part of this pattern and saves it in a pattern register; "DRAW" is a mode in which the user can modify the pattern by moving the mouse, as it were a pen or an eraser. The other entries: "ROTATE", "REFLECT" and "DRAW LINE" have not been implemented yet.

5. grey menu: menu lying just under the previous one, and proposing various commands to modify the patterns displayed in the window "pattern_window": "CLEAR GREY" kills the pattern displayed in grey; "MOVE GREY" translates this pattern; "ADD" adds the pattern displayed in grey to the pattern displayed in black; "SWAP" exchanges the pattern displayed in grey and the pattern displayed in black.

6. parameters display menu: menu lying just under the previous one, and displaying the values of three parameters of a character pattern: its width, its height and its bias. They correspond to the width, the height and the distance between the middle line and the bottom line of the box displayed in the window "pattern_window". Currently its display is not updated in real time. When the user changes one of these parameters by modifying the shape of the box, the new values of the parameters are not automatically displayed. This should be

fixed. They parameters displayed are only updated when the user enters the command "REAL SIZE".

7. pattern_window: biggest window of the screen, is in charge of displaying two patterns, one in grey and one in black, in such a way that the dots of the patterns appear as squares of several dots wide. It also displays a box which gives limits to the dimensions of the patterns. During edition, the user can change the dimensions of this box, and write parts of the patterns outside of it. But only the parts of the patterns lying inside the box will be sent to the character mode for building new characters.

8. registers_window: horizontal window at the top of the screen in charge of handling pattern registers and of displaying their contents on its screen.

9. sample_window: small window at the bottom left corner, in charge of displaying patterns in real size. Is only used through the commands "REAL SIZE" and "SET SAMPLE" of the menu "global_menu".

10. left_margin:is first set to the maximum width of all the menus of this mode, and slightly increased afterwards if necessary to ensure that the width of the window "pattern_window" is a multiple of the grid_size. Used for computing the definitive sizes and positions of the windows and menus.

11. upper_margin: corresponds to the height of the window "registers_window". Its value is such that the height of the window "pattern_window" is a multiple of the grid size, and the height of each register box in the window "registers_window" is at least equal to the height in number of squares of the window "pattern_window".

12. lower_margin: is set to the sum of the heights of all the menus, and used for computing the size and the position of the window "sample_window".

13. grid_size: is the size of the squares used in the window "pattern_window" to represent dots. The default value is 20, the admissible values run from 5 to 25. Values smaller than 5 can hardly be used; values greater than 25, up to 40 or 50, could probably be used. However, it should be checked if they are really useful or not before allowing their use.

14. number_of_registers: is set to the largest number of registers the window "registers_window" can display in such a way that each register box is wide enough to display the full contents of the window "pattern_window".

3.5.3 Instance Methods

:menus_list(Instance,^List_of_menu_names)
(See the method menus_list of the class char_mode_display).

:windows_list(Instance,^List_of_window_names)
(See the method windows_list of the class char_mode_display).

:items_list(Object,Menu_name,^Items)
(See the method items_list of the class char_mode_display).

:parameters_list(Instance,Window_name,^Parameters_list)
(See the method parameters_list of the class char_mode_display).

:get_grid_size(Instance,^Grid_size)
:set_grid_size(Instance,Grid_size)
Checks if the new grid size Grid_size has an admissible value (integer between 5 and 25); if not, changes its value to be equal to 5 or to 25; then if this new grid size happens to be equal the old one, does nothing; otherwise, saves this new grid size in the slot "grid_size", creates and shows a full screen window to hide what part of the transitional states of the display. Calls the methods :reshape, :show and :draw which are inherited from the class as_font_display, and kills the full screen window.

before:draw(Instance)
The :draw method is inherited from the class as_font_display. This before demon sends to the window "pattern_window" the current value of the grid size, and to the window "registers_window" the current value of the slot "number_of_registers". The :draw method is not used very often, probably only at the creation of the display, and each time the grid size is modified.

3.5.4 Class Method

after:create(Class,^Instance,Superior_window)
The :create method is inherited from the class as_font_display. The after demon calls another method inherited from this class, the instance method :initialize. This demon, as well as the equivalent one of the class char_mode_display, should be replaced by adding the goal :initialize to the :create method itself.

3.5.5 Local Predicates

set_margins(Instance)
This method must be used just after the menus have been created. Computes the two margins "left_margin" and "lower_margin" from the minimum sizes the window system gave

to the menus when they were created, and, at the same time, sets the positions of the menus; then adjusts the margin "left_margin" so that the width of the window "pattern_window" is a multiple of the grid size, computes the margin "upper_margin", and the parameter "number_of_registers", as indicated above; then adjusts the widths of the menus to be equal to the value of the slot "left_margin".

set_margins_and_positions(Instance,List_of_menus_names,Position)
Removes from the list List_of_menus_names the next menu name, gets this menu (the name of which is nothing else than the name of the slot in which it has been saved), asks for its size, sets its position to be (0,Position), modifies the slot "left_margin" if its current value is smaller than the width of this menu, adds the height of this menu to the pointer Position, and calls itself recursively. When the list becomes empty, sets the value of the slot "lower_margin" to be equal to the value of the pointer Position, and stops. At that moment, Position is equal to the sum of the heights of the menus. The initial value of the pointer must be 0.

adapt_margins_to_grid_size(Instance)
Adds to the present value of the slot "left_margin" just what is necessary for the integer (screen_width - left_margin) to be a multiple of the integer grid_size.
Determines the value of the slot "number_of_registers" in such a way that each register could be allocated just enough space on the screen of the register window to display the contents of the window "pattern_window". The 6 that appears in the formula comes from the fact that the frames delimiting the regions allocated to different registers must not be considered as an usable part of the display. The width of the frame is 3.
Computes an intermediate variable, Upper_margin, in such a way that each register will be allocated just enough space on the screen of the register window to display the contents of the window "pattern_window". Then adds to that variable what is necessary for assuring that the height of the window "pattern_window" will be a multiple of the grid size.

set_sizes(Instance,List_of_menus_names)
Sets the widths of the menus the name of which appears in the list List_of_menus_names to the value "left_margin".

menus_list(List_of_menus_names)

windows_list(List_of_windows_names)

items_list(Menu_name,(Class_name,Items_list))
Items_list respects the format of the menu items of the window system, except for the menu "parameters_display_menu", which has a special format adapted to the specific needs of the font editor. See the class parameters_display_menu for details.

parameters_list(Instance,Window_name,(Class_name,
[position(X,Y),size(Width,Height)]))

(See the method parameters_list of the class char_mode_display).

check_grid_size(Instance,Input_grid_size,Grid_size)
Input_grid_size is supposed to be an integer.

CHAPTER 4

GRAPHIC_INTERFACE

4.1 DESIGN CONSTRAINTS

It has been seen in the previous chapter that the font editor needs various kinds of windows and menus.

1. First of all, windows displaying character patterns in real size are required. Methods for drawing a pattern must allow the programmer to precise the position of the pattern in a window, as well as the part of the pattern or the part of the window that must be used.

2. Then standards for patterns must be fixed, and windows must be able to send and receive information, not only patterns but also all the relevant parameters that must be used to ensure the expected functionalities. Editing facilities will be treated in the next chapter.

3. Moreover, windows for displaying various kinds of parameters must be provided, as well as methods for accessing these parameters.

4.2 DESIGN COMMITMENTS

At that level, there is not too many choices for the design of the classes. Mainly, what has been decided is to create small classes adapted to each basic functional requirement of the overall design. It means up to ten different windows or menus classes which fit some special application, instead of two or three general classes.

A basic requirement of the top level objects is that all the windows and menus may be read from a superior window of the size of the screen. Although this facility is provided by the window system, it has not been introduced in usual windows and menus classes; it is the main reason why the font editor needs special windows and menus classes, namely: font_inferior_window, font_superior_window, font_multiple_select_window, font_multiple_select_multi_column_window.

Graphics functions, since they write directly in the bitmap

memory, must be executed through the window manager. To allow easier modifications of the interface between the window manager and the font editor, it has been decided to create a window class: the class font_graphics_window, in which the methods that must be executed inside the window manager are regrouped.

The standard format used for storing and transmitting patterns as messages between objects will be instances of the class char_record_with_pattern.

The character mode display needs windows that can display a pattern, and behaves like a register. Instances of the class pattern_display_window have been designated for meeting this needs.

The pattern mode display needs a window that can display a line of characters of the font currently edited. This window will be an instance of the class display_sample_window.

A requirement a little more complex from the pattern mode display is a window that can display several patterns, each in a box, in such a way that the boxes seem to behave like pattern registers for the user. This window will be an instance of the class pattern_registers_window.

Another requirement, from the character mode display this time consists of a window that can handle all the patterns of a font, display some of them, scroll, and allow direct access to each of these patterns. this window will be an instance of the class font_display_window.

The last requirement of the character mode display consists of two simple windows used for displaying the codes of the characters. These windows should be able to display a list of integers, to scroll, and to offer the possibility of changing the distance between two consecutive integers. These windows will be instances of the classes horizontal_code_display_window and vertical_code_display_window.

The last requirement of the pattern mode display consists of a simple window displaying three parameters. This window is implemented as a menu, and will be an instance of the class parameters_display_menu.

4.3 THE CLASS FONT_INFERIOR_WINDOW

4.3.1 Description

This class is almost entirely built by inheriting classes provided by the window system. It inherits the following classes: with_superior_input_buffer, which allows inputs done through its instances to be read from a superior window; as_input; as_mouse_input; as_output; as_graphics; user_window, which is a basic window class. For more details on these classes, see the window system manual.

4.3.2 Instance Attribute

An instance of this class has 1 attribute.

1. title: it is a fixed attribute, in which the name of the window is stored at the creation (see the class as_font_display). The usual attribute "title" of windows cannot be used here, because the class which provides it, the class with_label has not been inherited. However, the choice of the name of this attribute may be confusing. Another name for it as well as for the methods :get_title and :set_title should be considered.

4.3.3 Instance Methods

:get_title(Instance,^Title)

:set_title(Instance,Title)

4.3.4 Class Method

after:create(Class,Parameters_list,~Instance)
Sets the inside units of the window to the values (1,1). The default inside units given by the window system are the width and the height of the space character in the default font font_13, and are not suitable for graphics. Sets the permission of the window to "out", so that it is possible to write into this window even when it is not completely shown.

4.4 THE CLASS FONT_SUPERIOR_WINDOW

4.4.1 Description

This class is entirely built by inheriting classes provided by the window system. It inherits the following classes: as_input_buffer, which allows the inputs done through an inferior window to be read from this window; as_superior, user_window. For more details on these classes, see the window system manual.

4.5 THE CLASS FONT_MULTIPLE_SELECT_MENU

4.5.1 Description

This class is almost entirely built by inheriting classes provided by the window system. It inherits the following classes: sash, which provides a frame delimiting the area on the screen used by the menu; with_superior_input_buffer, which allows the inputs done through this menu to be read from a superior window; as_menu; as_multiple_select; as_mouse_input; user_window, which is a basic window class. For more details on these classes, see the window system manual.

4.5.2 Instance Attribute

An instance of this class has 1 attribute.

1. title: it is a fixed attribute, in which the name of the menu is stored at the creation (see the class as_font_display and the class font_inferior_window).

4.5.3 Instance Methods

:get_title(Instance,^Title)

:set_title(Instance,Title)

4.6 THE CLASS FONT_MULTIPLE_SELECT_MULTI_COLUMN_MENU

4.6.1 Description

This class is almost entirely built by inheriting classes provided by the window system. It inherits the following classes: sash, which provides a frame delimiting the area on the screen used by the menu; with_superior_input_buffer, which allows the inputs done through this menu to be read from a superior window; as_multiple_column; as_multiple_select; as_mouse_input; user_window, which is a basic window class. For more details on these classes, see the window system manual.

4.6.2 Instance Attribute

An instance of this class has 1 attribute.

1. title: it is a fixed attribute, in which the name of the menu is stored at the creation (see the classes as_font_display and font_inferior_window).

4.7 THE CLASS FONT_GRAPHICS_WINDOW

4.7.1 Description

This class inherits the class font_inferior_window. Its main role is to provide to classes inheriting it methods for drawing a pattern in a given area of a window, cutting what must not be displayed. It is one of the classes which have methods that must be executed inside the window manager. The other one is the class as_edit_pattern_window.

It is not possible in a multi-task machine to allow a direct access to the bitmap memory to processes. The access must be controlled somehow, and, in the present version of the PSI operating system, this task has been attributed to the window manager. It is why the methods of the classes font_graphics_window and as_edit_pattern_window which use bitmap areas for storing patterns and transferring them on window screens must be currently executed inside the window manager.

4.7.2 Instance Attributes

1. operation: is the operation used by the methods :draw_record/4 and :draw_record/8 to draw a pattern in the window. Its default value is 'exclusive_or'; it can be otherwise set to 'inverse' or 'copy'. In the current version, the font editor uses only the default value 'exclusive_or'.

2. position: is initialized to the value 'inside'. The other possible value is 'outside'. This slot is used when a pattern has to be drawn. In the case the pattern is entirely outside the window, the slot position is set to the value 'outside', otherwise to 'inside'. When the value of this slot is 'outside', the drawing routine is not called.

3. window_x: The methods :draw_record do not display the part of a pattern which horizontal coordinate is smaller than the value of this slot.

4. window_y: The methods :draw_record do not display the part of a pattern which vertical coordinate is smaller than the value of this slot.

5. box_height: The methods :draw_record cannot display entirely a pattern of height greater than the value of this slot, and must cut a part of it.

6. box_width: The methods :draw_record cannot display entirely a pattern of width greater than the value of this slot, and must cut a part of it.

4.7.3 Instance Methods

:draw_record(Instance,Record,X0,Y0)
The variable Record must be instantiated to an instance of the class char_record_with_pattern. This method draws the pattern of the record Record at the position (X0,Y0) in the window. The position of the pattern is the position of the top left corner of the pattern. (X0,Y0) can lie outside the window: in that case, the pattern may be partially drawn, or not drawn at all. Any part of the pattern which would have appeared outside the window or at less than 3 dots from the edge of the window is not drawn.

:draw_record(Instance,Record,X0,Y0,Limit_x,Limit_y,Width,Height)
The variable Record must be instantiated to an instance of the class char_record_with_pattern. The parameters Limit_x, Limit_y, Width and Height define a box. Limit_x and Limit_y are the coordinates of the top left corner of this box, while Width and Height are its width and height. This method draws the pattern of the record Record at the position (X0,Y0). The parts of the pattern which would have lain outside the window or outside the box are not displayed.

:do_draw_record(Instance,Record,X0,Y0)
This is the method used inside the window system. Draws the pattern of the record Record at the position (X0,Y0), cutting the part of the pattern which lies outside the window or outside the rectangle of width "box_width" and height "box_height" and of position ("window_x","window_y"). These four attributes are initialized by the methods :draw_record before the method :do_draw_record is called.

:get_operation(Instance,^Operation)

:set_operation(Instance,Operation)

4.7.4 Class Method

after:create(Class,List_of_parameters,~Instance)
This after demon erases the undesirable cursor of the window.

4.7.5 Local Predicates

draw_record(Instance,Record,X0,Y0)
If the flag "position" is set to 'outside', does nothing. Otherwise creates a bitmap area, loads the pattern of the record Record in it, copies this area or a part of it in the window screen, according to the diverse restrictions already mentioned, and kills the area.

open_area(Instance,Record,~Area,^Area_width,^Area_height)
Asks to the record its size; instantiates the variable

Area_height to the height of the record; asks to the record its pattern, which is a double_bytes string; asks to this pattern its length, and divides this length by the height of the pattern. The result is the raster width of the pattern. Instantiates the variable Area_width to 16 times the raster width. Then allocates a bitmap area Area of size (Area_width,Area_height), and loads the pattern in this area.

transfer_area(Instance,Record,Area,Area_width,Area_height,X0,Y0)
Asks to the record its width; computes the difference between the Area_width and the record width, which corresponds to the number of empty vertical lines that had to be added to the pattern of the record when it was stored in a double_bytes string. Then computes which part of the area Area must be transferred.

The part to be transferred must not include any of the empty vertical lines. Moreover, when the limit "window_x" is greater than the position X0, the pattern must be cut on its left of window_x - X0 lines. Similarly, when the limit "window_y" is greater than the position Y0, the pattern must be cut at its top of window_y - Y0 lines.

The width of the part transferred is limited both by the width of the area after the previous operations and by the value of the attribute "box_width". The height of the part transferred is limited both by the height of the area after the previous operations and by the value of the attribute "box_height".

If, after these operations, there is nothing to display, does nothing; otherwise transfers the selected part of the area into the window screen, at the position ("window_x","window_y").

set_limits(Instance,X0,Y0,Limit_x,Limit_y,Width,Height,Margin)
Increases Limit_x and Limit_y by Margin, Width and Height twice by Margin, and calls the local predicate set_limits/7.

set_limits(Instance,X0,Y0,Limit_x,Limit_y,Width,Height)
Takes the greatest integer between X0 and Limit_x, checks if the resulting horizontal coordinate can be a the horizontal coordinate of a dot lying inside the window; does the same for the vertical position; if it is the case for both, sets the values of the slots "window_x", "window_y", "box_width" and "box_height" to be the corresponding values sent as parameters of this predicates; otherwise sets the flag "position" to the value "outside".

4.8 THE CLASS PATTERN_DISPLAY_WINDOW

4.8.1 Description

This class inherits the class sash, which provides a frame delimiting the area on the screen used by the window, and the class font_graphics_window, which provides methods for writing a pattern in the window screen. An instance of this class behaves like a register for patterns. A pattern stored in a window of this class is saved in an attribute. It is supposed to be in the standard format of instances of the class char_record_with_pattern. A window of this class is destined for communicating with the user in the following way: when the user wants to load a pattern in it, he selects a pattern in the window displaying the character patterns of a font, and then click the mouse in the window of this class; when he wants to do the opposite, he selects first this window, and then the destination of the pattern. For allowing the latter, an instance of this class has another attribute for remembering whether or not the user has selected it.

4.8.2 Instance Attributes

1. status: remembers if this window has been already selected. Its initial value is 'nil'; when the window is selected, its value is 'select'. Can be reset to 'nil' from outside simply by selecting this window one more time. This selection is used by the font editor for remembering inputs from the user, and has nothing to do with the selection of windows in the sense of the window manager. It may be better to change the name of this attribute and of the corresponding methods from 'select' to 'activated'.

2. record: is a slot for saving an instance of the class char_record_with_pattern. If no record is saved in it, its value is set to the atom 'nil'.

4.8.3 Instance Methods

:get_record(Instance,~Record)

:set_record(Instance,Record)

:select(Instance)
If the flag "status" is set to 'nil', sets it to 'select'; otherwise sets it to 'nil'.

:get_selected_status(Instance,^Status)

:draw_display(Instance)
clears the display of the window; if the slot "record" is set to 'nil', does nothing else; otherwise, the slot "record" contains a record; draws this record, centered in the middle of the window screen.

4.8.4 Local Predicate

center_pattern(Instance,^X0,^Y0)
computes the position that the pattern of the record "record" must have to be displayed in the middle of the window screen.

4.9 THE CLASS DISPLAY_SAMPLE_WINDOW

4.9.1 Description

This class inherits the class sash, which provides a frame delimiting the area on the screen used by the window, and the class font_graphics_window, which provides methods for writing a pattern in the window screen. An instance of this class can display patterns, recorded in a list of char_record_with_pattern instances. It has two modes of display: the 'sample' mode, in which several patterns are displayed on one or several lines, and the 'real size' mode, in which only one pattern is displayed. This distinction is not really relevant, and may be suppressed with profit. The origin of the present state is that the method for displaying one pattern has been implemented first, and the method for displaying several patterns later.

4.9.2 Instance Attributes

1. record: is a slot for saving an instance of the class char_record_with_pattern. If no record is saved in it, its value is set to the atom 'nil'. In 'sample' mode, the contents of the slot, if different form the atom 'nil', are added to the list of records to be displayed. In 'real size' mode, this record, if different from the atom 'nil', is displayed.

2. line_height: is computed each time a new set of records must be displayed in the 'sample' mode. Its value is equal to the maximum apparent height of the given set of records, increased of one third to leave some space between lines.

3. sample: is a flag which keeps the information about the current mode. May take two values: 'yes' if the current mode is 'sample', 'no' if the current mode is 'real size'.

4. record_list: is an instance of the class list. In both the 'real size' and the 'sample' mode, the record or the records to be displayed are stored in this list, which is read afterwards by the display routine.

4.9.3 Instance Methods

:set_record(Instance,Record)
Saves the record Record into the slot "record"; if the record Record is equal to 'nil', does nothing; otherwise, clears the list "record_list", and saves the record Record in it.

:set_sample(Instance,List_of_records)
List_of_records must be an instance of the class list, and must contain instances of the class char_record_with_pattern. Saves the list List_of_records into the slot "record_list". If the contents of the slot "record" are different from the atom 'nil', adds them to the list "record_list". Then computes the highest and the lowest positions that the patterns of the records stored in the list "record_list" take on a character line; then saves their difference, increased by one third, into the slot "line_height". At last sets the 'sample' mode.

:draw_display(Instance)
clears the display of the window; in 'sample' mode, takes the contents of the list "record_list", displays the patterns of the records contained in this list one after the other, stopping if no more space is available in the window, and switches back to the 'real size' mode. In 'real size' mode, checks if the slot "record" does not contain the atom 'nil': if it is the case, does nothing; otherwise, displays the record Record.

4.9.4 Local Predicates

draw_display(Instance,List_of_records,X0,Y0)
Removes the next record from the list List_of_records, computes its position (X1,Y1), from (X0,Y0), by adding to X0 one sixth of the width of the record, or by going to the next line; checks if any space is available in the window to draw the pattern of the record; if not, stops; otherwise draws the pattern, and calls itself recursively, with new values for (X0,Y0) to be (X1,Y1) modified by increasing X1 of seven sixths of width of the pattern.

compute_position(Instance,Record,X0,Y0,^X1,^Y1,^Message)
Checks if X0 increased by the width of the pattern of the record Record is less than the width of the window; in that case, the pattern can be drawn on the current line; so sets X1 to be equal to X0 increased by one sixth of the width of the pattern, to leave some space between characters. Otherwise, checks if Y0 increased by the "line_height" is less than the height of the window: in that case the pattern can be drawn on the next line; so sets (X1,Y1) to be the position of the next character on the next line. In these two cases, Message is instantiated to the atom 'inside'. Otherwise, Message is instantiated to the atom 'outside'.
The case when the first pattern to be drawn on a line is

wider than the window is not treated consistently. If it happens, the pattern will just be cut to fit the dimensions of the window, instead of not being displayed at all.

compute_max_height(Instance,List_of_records,^Max_height)
Computes the maximum height of the patterns of the records contained in the list List_of_records, as well as their maximum bias, and instantiates the variable Max_height to the sum of these two maximums.
This algorithm has a logical bug: it does work, does not prevent this window to display usual patterns, but does not compute the parameter it is supposed to compute. The parameter it is expected to compute is the maximum distance between the top of the highest pattern of the list and the bottom of the lowest. The error is that it is not the maximum height that must be added to the maximum bias, but the maximum value of the difference (height - bias).

compute_max_height(Instance,Records_list,^Max_height,^Max_bias)
Computes the maximum height of the patterns of the records containing in the list Records_list, as well as their maximum bias, by using a recursive procedure.

4.10 THE CLASS PATTERN_REGISTERS_WINDOW

4.10.1 Description

This class inherits the class sash, which provides a frame delimiting the area on the screen used by the window, and the class font_graphics_window, which provides methods for writing a pattern in the window screen. An instance of this class can keep several patterns in registers and display their contents.

At each register corresponds a rectangular part of the window screen, called a box, in which the contents of the register are displayed, and through which the contents of the register can be accessed by the user directly by clicking the mouse in it. Each of these boxes is big enough to display the contents of the window used for editing patterns.

Since the size of the boxes roughly correspond to the size in number of squares of the editing window, and since the width of this window is roughly fixed, the number of boxes, and therefore the number of registers may vary considerably from one grid size to another. When the grid size is decreased, the number of registers is also decreased, and, thus, some patterns may be lost in the operation. if too many registers were used, the contents of some of them are lost.

The box on the left of the window has a special role. It does not correspond really to a register, but to a stack. This stack is called the protected register, because it is used for saving patterns that would have been thrown away otherwise, and therefore must protected to be effective as a protection.

When the grid size is decreased, the patterns that could be lost otherwise should be saved in this stack.

4.10.2 Instance Attributes

1. number_of_registers: the number of registers of this window is computed by an instance of the class pattern_mode_display each time a new grid size is set, and then sent to this window and saved in this attribute.

2. first_column: The register boxes cannot usually have the same width for an obvious arithmetic reason. This attribute contains the width of the first box, computed in such a way that the other boxes may have the same width.

3. espacement: This attribute contains the width of all the boxes, with the possible exception of the first one.

4. height: this attribute contains the height of this window. May not be very useful since it is always possible to ask the height of a window by the method :get_size, provided by the window system. To suppress this attribute and to use the window system method each time the height is required is probably better.

5. pile: The name of this attribute, as the word "espacement", comes from the French. The English equivalent of "pile" is "stack". Is an instance of the class list, and contains all the records stored in the first register, except the one which is displayed, which is stored in the table "table_of_contents".

6. table_of_contents: Is an instance of the class hash_index, and contains all the contents of the registers of this window that are displayed. The registers are numbered from 0 to "number_of_registers" - 1, from left to right. The key used for storing the contents of a register into the hash_index is the number corresponding to this register.

4.10.3 Instance Methods

:get_number_of_registers(Instance,^Number_of_registers)

:set_number_of_registers(Instance,Number_of_registers)
sets the contents of the slot "number_of_registers" to be the value of the variable Number_of_registers, and saves the old value in a local variable; computes the width of window screen which can be allocated to the register boxes, in such a way that all the boxes have the same width with the possible exception of the first one, which may be given a little more space, but not enough to be distributed equally among the

other boxes. Then creates a new instance of the class hash index , saves in it the contents of the old hash_index as long as there are enough registers left for displaying them, and saves the new hash index in the slot "table_of_contents".

:draw_display(Instance)
clears the display of the window, draws the contents of each register, and draws the lines delimiting the register boxes. In the case of the first register, which can save several patterns in its stack structure, only the last entered pattern is displayed.

:draw_contents(Instance)
Checks if the table "table_of_contents" is empty. If it is the case, does nothing. Otherwise, draws its contents, each pattern centered in the corresponding box.

:get_status(Instance,X,Y,^Status)
(X,Y) corresponds to a position of the mouse. This predicate computes in which register box the user clicked the mouse; if it is the first register, instantiates the variable Status to the atom 'protected'; otherwise, checks if the selected register is empty; if it is empty, instantiates Status to the atom 'empty'; if it is not empty, to the atom 'full'.

:save_into_register(Instance,X,Y,Record)
Computes from the position (X,Y) in which register box the user clicked the mouse; if this register is the first one, does nothing; otherwise saves the record Record into this register, and modifies the display so that the pattern of the record Record is displayed in the box of this register.

:save_into_protected_register(Instance,Record)
If the first register is empty, saves the record Record into the table "table_of_contents" at the key 0. Otherwise, replaces in the table "table_of_contents" the previous record saved at the key 0 by the record Record, and adds this previous record at the front of the list "pile". In both cases, modifies the display so that the pattern of the record Record is displayed in the first box.

:get_register_contents(Instance,X,Y,~Record)
Computes from the position (X,Y) in which register box the user clicked the mouse; then gets from the table "table_of_contents" the record Record at the key corresponding to the selected register. In the case the selected register is the first register, replaces at the key 0 in the table "table_of_contents" the record Record by the first record at the front of the list "pile", if any, and modifies the display of the first box if necessary.

:remove_register_contents(Instance,X,Y,Record)
Computes from the position (X,Y) in which register box the user clicked the mouse; removes from the table "table_of_contents" the record stored at the key corresponding to the selected register, and clears the box of this register.

This does not work properly in the case of the first register, since the contents of the list "pile" are not either cleared or used. This is another logical bug. Before correcting it, it must be checked if this method is used or not in that case.

:clear_registers(Instance)
clears the display and the contents of all the registers.

4.10.4 Local Predicates

draw_frame(Instance)
draws the vertical lines separating the register boxes.

draw_vertical_lines(Instance,Initial_x,Espacement,
Height,Number_of_lines,N)
draws a vertical line of thickness 4 at the position Initial_x through all the height of the window, increases Initial_x of Espacement, N of 1, and calls itself recursively. Stops when N becomes equal to Number_of_lines.

draw_contents(Instance)
(see the method :draw_contents)

draw_contents(Instance,Table,Number_of_registers,N)
tries to get a record saved at the key N in the table "table_of_contents"; if there is one, draws it, centered in the box number N; if there is none, clears the box number N; increases N by 1 and calls itself recursively. Stops when N grows bigger than or equal to Number_of_registers.

draw_pattern(Instance,Record,N)
draws the pattern of the record Record in the middle of the box number N. The pattern is cut to fit the dimensions of the box if necessary. The case of the first register is treated apart only because the width of its box is not the same than the width of the other register boxes.

clear_register_display(Instance,N)
Clears the box number N. The case of the first register is treated apart only because the width of its box is not the same than the width of the other register boxes.

get_status(Instance,X,Y,^Status)
Computes from the position (X,Y) in which register box the user clicked the mouse; then determines the status of this register. If it is the first register, the status is 'protected'; otherwise, if the register is empty, the status is 'empty'; otherwise the status is 'full'.

get_status(Instance,N,^Status)
determines the status of the register number N, as indicated for the local predicate get_status/4.

get_register_number(Instance,X,Y,^N)

Computes the number N of the box in which the user clicked the mouse. (X,Y) is the position of the mouse just after the clicking.

save_into_register(Instance,X,Y,Record)
(see the method :save_into_register. Note that the draw_pattern predicate uses an exclusive_or operation, and that calling it when a pattern is already displayed erases it).

save_into_protected_register(Instance,Record)
(see the method :save_into_protected_register).

get_register_contents(Instance,X,Y,~Record)
(see the method :get_register_contents).

get_contents(Instance,N,Status,~Record)
Gets the contents of the register number N, and instantiates the variable Record with them; if the register number N is empty, instantiates Record to the atom 'nil'. In the case of the first register, replaces at the key 0 in the table "table_of_contents" the record Record by the first record at the front of the list "pile", if any, and modifies the display of the first box if necessary.

remove_register_contents(Instance,X,Y,~Record)
(see the method :remove_register_contents).

try_to_save(Old_table,Table,Old_number,Number)
saves the contents of the table Old_table in the table Table, as long as the table Table, of size Number, is not full.

try_to_save(Old_table,Table,Old_number,Number,Old_pointer,Pointer)
if there is a record saved at the key Old_pointer in the table Old_table, saves it in the table Table at the key Pointer, increases Old_pointer and Pointer by 1, and calls itself recursively; otherwise, increases Old_pointer by 1 and calls itself recursively. Stops when either Old_pointer is equal to Old_number or Pointer is equal to Number.

4.11 THE CLASS FONT_DISPLAY_WINDOW

4.11.1 Description

This class inherits the class sash, which provides a frame delimiting the area on the screen used by the window, and the class font_graphics_window, which provides methods for writing a pattern in the window screen. An instance of this class is in charge of displaying the character patterns of a font. The characters are displayed according to their codes: on the same line, the characters the codes of which have the same 8 higher bits; on the same column, the characters the codes of which have the same 8 lower bits. Scrolling facilities are provided so that the user can choose which characters he wants to see

displayed.

Other facilities, as the possibility to select a character code by clicking the mouse at the intersection of a row and a column, or to create a new character by simply selecting a pattern, taken from the pattern mode or from another character, and assigning it to a code, are also provided.

Moreover, each time a new pattern is introduced as a new character pattern, the distances between two consecutive rows or two consecutive columns are checked and if necessary, modified, to avoid overlaps.

4.11.2 Instance Attributes

1. font: contains either the atom 'nil' or an instance of the class edited_font. It is the font which has its characters displayed in this window.

2. selected code: contains the code selected if there is one, or, otherwise, the atom 'nil'.

3. default code: is a fixed attribute, initialized at the value 0. Each time the display of the window is drawn, the pattern of each code displayed is drawn; if there is none, the pattern corresponding to the default code is drawn; if there is no pattern corresponding to the default code, nothing is done. It is up to the user to decide to assign a default pattern to the default code. The usefulness of the default pattern has to be determined by the user.

4. initial code: is the value of the smallest code displayed. It is only used for avoiding to recompute its value each time it is required. It is not absolutely indispensable.

5. initial code x: is the value of the lower 8 bits of the smallest code displayed.

6. initial code y: is the value of the higher 8 bits of the smallest code displayed.

7. displayed x: is the number of columns displayed.

8. displayed y: is the number of rows displayed.

9. espacement x: is the distance between two consecutive columns displayed.

10. espacement y: is the distance between two consecutive rows displayed.

11. initial x: is the position of the left end of the first column displayed in the window.

12. initial_y: is the position of the top end of the first row displayed in the window.

4.11.3 Instance Methods

:get_operation(Instance,^Operation)
This method is no longer in use. Must be suppressed.

:set_operation(Instance,^Operation)
This method is no longer in use. Must be suppressed.

:get_espacement(Instance,^Espacement_x,^Espacement_y)

:set_espacement(Instance,Espacement_x,Espacement_y)
saves Espacement_x and Espacement_y in the corresponding attributes; computes, from their values and the size of the window, the number of rows and columns that can be displayed, and saves these values into the slots "displayed_x" and "displayed_y".

:set_font(Instance,Edited_font)

:get_font(Instance,^Edited_font)

:reset_espacement(Instance)
If there is no font to display, resets the slots "espacement_x" and "espacement_y" to 0; otherwise, asks to the font "font" its global raster width and the maximum height its characters use on a line, taking their bias into account; sets the slot "espacement_x" to be 16 * raster width increased by 50 per cent, and "espacement_y" to be the maximum height increased by 50 per cent, and recomputes the number of lines and rows that can be displayed in the window.

:draw_display(Instance)
clears the display of the window, if there is no font to display, and if no code has been selected, stops; if one code has been selected, and if this code is displayed, draws a rectangle of the size of a character at the position corresponding to this code, and stops. If there is one font to display, for each displayed code, draws the corresponding pattern, if any; if there is none, draws the pattern corresponding to the default code, if any; if there is none, does nothing. Then checks if one code has been selected; in that case, and if the code is displayed, draws a rectangle at the position corresponding to this code.

:scroll(Instance,Message)
Message can be one of the following atoms: 'left', 'right', 'up', 'down'. Scrolls each time of one page, except when no more columns or no more rows have to be displayed.

:get_selected_code(Instance,^Selected_code)

Answers 'nil' if no code is currently selected.

:select_code(Instance,X,Y)
computes the code which has been selected by the user by a mouse click with the mouse at the position (X,Y), and calls the method :select_code(Instance,Code).

:select_code(Instance,Code)
draws a rectangle at the position corresponding to the previously selected code, if there is one, and if it is displayed. Since the rectangle is drawn using an 'exclusive_or' operation, it has for effect to erase the rectangle previously drawn. Then checks if the code Code is equal to the previously selected code, if any; if it is the case, forgets this code, and puts the atom 'nil' into the slot "selected_code"; otherwise, puts the new code Code into the slot "selected_code", and draws a rectangle around the new Code if it is displayed.

:draw_char(Instance,Code)
if the variable Code is instanstiated to the atom 'nil', does nothing. If the code Code is not currently a displayed code, does nothing; otherwise, computes the 8 lower bit part and the 8 higher bit part of the code Code, and uses them as coordinates for drawing the pattern associated to the code Code, or, if there is none, the default pattern. If there is no default pattern either, does nothing.

4.11.4 Local Predicates

draw_display(Instance)
draws on the screen the patterns corresponding to the displayed codes. Moreover, if a pattern has been assigned to the default code, draws it as the corresponding pattern of all the codes which do not have their own corresponding patterns.

draw_display(Instance,Displayed_y,Y)
draws the patterns that must be displayed in the row number Y, increases Y by one and calls itself recursively. Stops when Y is equal to Displayed_y, the number of rows displayed.

draw_line(Instance,Y,Displayed_x,X)
draws the pattern which must be displayed at the intersection of the row number Y and the column number X, increases X by one and calls itself recursively. Stops when X is equal to Displayed_x, the number of columns displayed.

draw_char(Object,X,Y)
Computes the code corresponding to the intersection of the row number Y and the column number X. Gets the record associated to this code; if there is none, the record associated to the default code; if there is none, stops. Otherwise, displays the pattern of the record, centered in the middle of the

intersection zone between the row number Y and the column number X.

get_record(Instance,~Record,Code)
gets the record, an instance of the class char_record_with_pattern, corresponding to the code Code, in the font "font". If there is none, gets the record associated with the code "default_code". If there is none, instantiates Record to the atom 'nil'.

scroll(Object,Message)
Computes the next values of the slots "initial_code", "initial_code_x" and "initial_code_y" according to the direction of scroll required by the message Message, clears the display and draws it again, displaying codes according to the new values of these slots.

new_code(Instance,Message)
computes the new codes to be displayed. Message can instantiated to one of the following atoms: 'left', 'up', 'down' and 'right'. In the case where Message is instantiated to 'left', modifies only the slot "initial_code_x", by increasing it by the number of columns that can be displayed in one window screen, as long as there are any columns to be displayed left; modifies also the value of the slot "initial_code" consistently. The other cases are treated similarly.

next_code(Code,Number_of_codes,^New_code)
used by new_code, when "initial_code_x" or "initial_code_y" has to be increased.

previous_code(Code,Number_of_codes,^New_code)
used by new_code, when "initial_code_x" or "initial_code_y" has to be decreased.

compute_code(Instance,X,Y,^Code)
Here, (X,Y) refers to the position of the mouse on the screen. Computes the code Code corresponding to the the column and the row pointed by the mouse.

draw_code_box(Instance)
If no code is currently selected, does nothing. Otherwise, checks if the currently selected code is displayed or not; if it is not displayed, does nothing and stops; if it is displayed, computes the 8 lower bit and the 8 higher bit parts of the selected code, and calls the local predicate draw_code_box(Instance,Code_x,Code_y). Code_x corresponds to the lower 8 bit part of the code, Code_y to the higher 8 bit part.

draw_code_box(Instance,Code_x,Code_y)
draws a rectangle, using an exclusive_or operation, around the intersection zone of the row and the column corresponding to the values of Code_y and Code_x respectively.

4.12 THE CLASS HORIZONTAL_CODE_DISPLAY_WINDOW

4.12.1 Description

This class inherits the class sash, which provides a frame delimiting the area on the screen used by the window, and the class font_inferior_window, as a basic window class. An instance of this class displays integers in hexadecimal, from 0 to 255 (FF). These integers correspond to the 8 lower bits of the character codes of a font. Therefore, they correspond to columns of character patterns displayed in an instance of the class font_display_window.

All the numbers are not displayed together; are displayed only those which have their corresponding columns shown in the window displaying a font. The numbers which are displayed are centered above their corresponding columns. Since the distance between columns may change in the window displaying a font, the distance between numbers in this window must be changed accordingly. Methods are provided for handling this problem. Scrolling facilities are also provided.

4.12.2 Instance Attributes

1. initial_code: is the value of the smallest integer displayed. Corresponds to the slot "initial_code_x" of the window displaying a font.

2. string_width: is a fixed attribute, initialized to the width of a character string of length 2 in the window system default font "font_13". Absolute values are currently used instead of this slot value, which is not very good.

3. espacement: is the distance between two consecutive columns displayed in the window displaying a font. It is almost always the distance between two consecutive integers displayed in this window. The only exception comes from the fact that the integers are centered above their corresponding columns, and that their size can be of one character or of two. The exception occurs between "F" and "10".

4. number_of_codes_displayed

5. initial_x: is the horizontal coordinate of the first integer displayed in this window. Corresponds to the slot "initial_x" of the window displaying a font.

6. initial_y: is the vertical coordinate of all the integers displayed in this window. Has nothing to do with the slot "initial_y" of the window displaying a font.

4.12.3 Instance Methods

:get_espacement(Instance,^Espacement)
This method must be modified so that it does not answer the current value of the slot "espacement", but the minimum value the espacement can take in this window. See the class char_mode_display for details.

:set_espacement(Instance,Espacement)
saves the value of the variable Espacement in the slot "espacement"; computes the number of integers that it is possible to display, and initializes the slot "initial_y". It is not necessary to initialize the slot "initial_y" each time a new value is given to the slot "espacement". But it is necessary to do it at least once. It would be better that the initialization of the slot "initial_y" were done by the after demon after:create.

:draw_display(Instance)
clears the display of the window, and draws "number_of_codes_displayed" integers in it, in hexadecimal notation, beginning with the integer "initial_code".

:scroll(Instance,Message)
Message can be one of the following atoms: 'left' or 'right'. Scrolls each time of one page, except when no more columns have to be displayed. The scroll is coordinated with the scroll of the window which displays a font.

4.12.4 Class Method

after:create(Class,List_of_parameters,Instance)
erases the undesirable cursor, sets the espacement to the default value 35, and initializes the slots "initial_y" and "number_of_codes_displayed".

4.12.5 Local Predicates

draw_display(Instance)
draws the integers that must be displayed in the screen of the window. The integers are converted in strings of length 2, corresponding to their hexadecimal notation, and then displayed, centered in the interval of space which has been allocated to them. The width and the position of these intervals coincide with the width and the position of the columns associated with each integer in the window displaying a font.

draw_display(Instance,Buffer,Code,X,Y,Espacement,Width)
clears the buffer Buffer; using the method :writef of the class writef provided by the system, converts the integer Code into a string of length 2, corresponding to its hexadecimal

notation; the method :writef writes the resulting string into the buffer Buffer. Then computes the horizontal position of the string on the window screen from X, the horizontal coordinate of the left edge of the area reserved to the code Code in the window, and Espacement, the width of this area. There are two cases:

1- when the integer Code is smaller than 16, the string is of the form: " #", its horizontal position must be such that the "#" is centered in the interval [X,X + Espacement];

2- when Code is equal to or greater than 16, the string is of the form "##", and its horizontal position must be such that the "##" is centered in the interval [X,X + Espacement].

The vertical position of the string is simply Y, previously computed. Once the position is determined, draws the string, increases Code by 1 and X by Espacement, and calls itself recursively. Stops when X + Espacement becomes greater than or equal to Width, the interior width of the window.

scroll(Instance,Message)

Message can be either the atom 'left' or the atom 'right'. Computes, according to the message Message, the new value of the slot "initial_code", but keeps the old value for a while. Then draws one more time the old codes, in exclusive_or, which has the effect of erasing them, and draws the new codes instead. Then saves the new value of the slot "initial_code" in the slot itself. For aesthetic reasons, each time an old code is erased, a new one is written immediately afterwards.

redraw_display(Instance,Buffer,Code,New_code,X,Y,Espacement,Width)

Works like the local predicate draw_display/7, with the only difference that it writes two codes instead of one, at the same position: Code and New_code. Both operations are or exclusive operations. The first one erases the code Code, previously written, the second writes the new code New_code just afterwards, at the same position. Both codes must be centered according to their own values.

new_code(Code,Number_of_codes,Message,^New_code)

(See the local predicates new_code, next_code and previous_code of the class font_display_window).

4.13 THE CLASS VERTICAL_CODE_DISPLAY_WINDOW

4.13.1 Description

This class inherits the class sash, which provides a frame delimiting the area on the screen used by the window, and the class font_inferior_window, as a basic window class. It is very similar to the class horizontal_code_display_window. An instance of this class displays integers in hexadecimal, from 0 to 255 (FF). These integers correspond to the 8 higher bits of the character codes of a font. Therefore, they correspond to the rows of character patterns displayed in an instance of the class font_display_window.

All the numbers are not displayed together; are displayed only those which have their corresponding rows shown in the window displaying a font. The numbers which are displayed are centered at the left of their corresponding rows. Since the distance between rows may change in the window displaying a font, the distance between numbers in this window must be changed accordingly. Methods are provided for handling this problem. Scrolling facilities are also provided.

4.13.2 Instance Attributes

1. initial_code: is the value of the smallest integer displayed. Corresponds to the slot "initial_code_y" of the window displaying a font.

2. string_width: is a fixed attribute, initialized to the width of a character string of length 2 in the window system default font "font_13". Absolute values are currently used instead of this slot value, which is not very good.

3. espacement: is the distance between two consecutive rows displayed in the window displaying a font, and the distance between two consecutive integers displayed in this window.

4. number_of_codes_displayed

5. initial_x: is the horizontal coordinate of all the integers displayed in this window. Has nothing to do with the slot "initial_x" of the window displaying a font.

6. initial_y: is the vertical coordinate of the first integer displayed in this window. Corresponds to the slot "initial_y" of the window displaying a font.

4.13.3 Instance Methods

:get_espacement(Instance,^Espacement)

This method must be modified so that it does not answer the current value of the slot "espacement", but the minimum value the espacement can take in this window. See the class char_mode_display for details.

:set_espacement(Instance,Espacement)

puts Espacement in the slot "espacement"; computes the number of integers that it is possible to display, and initializes the slot "initial_x". It is not necessary to initialize the slot "initial_x" each time a new value for the slot "espacement" is set. But it is necessary to do it at least once. It would be better that the initialization of the slot "initial_x" be done by the after demon after:create.

:draw_display(Instance)
clears the display of the window, and draws "number_of_codes_displayed" integers in it, in hexadecimal notation, beginning with the integer "initial_code".

:scroll(Instance,Message)
Message can be one of the following atoms: 'up' or 'down'. Scrolls each time of one page, except when no more codes have to be displayed. The scroll is coordinated with the scroll of the window which displays a font.

4.13.4 Class Method

after:create(Class,List_of_parameters,Instance)
erases the undesirable cursor, sets the espacement to the default value 25, and initializes the slots "initial_x" and "number_of_codes_displayed".

4.13.5 Local Predicates

draw_display(Instance)
draws the integers that must be displayed in the screen of the window. The integers are converted in strings of length 2, corresponding to their hexadecimal notation, and then displayed, centered in the interval of space which has been allocated to them. The height and the position of these intervals coincide with the height and the position of the rows associated with each integer in the window displaying a font.

draw_display(Instance,Buffer,Code,X,Y,Espacement,Height)
clears the buffer Buffer; using the method :writef of the class writef provided by the system, converts the integer code into a string of length 2, corresponding to its hexadecimal notation; the method :writef writes the resulting string into the buffer Buffer. Then computes the vertical position of the string on the window screen from Y, the vertical coordirate of the upper edge of the area reserved to the code Code in the window, and Espacement, the height of this area. The vertical position of the string is computed to be such that it is displayed as centered in the interval [Y,Y + Espacement]. The horizontal position of the string is simply X. Once the position is determined, draws the string, increases Code by 1 and Y by Espacement, and calls itself recursively. Stops when Y + Espacement becomes greater than or equal to Height, the interior height of the window.

scroll(Instance,Message)
Message can be either the atom 'up' or the atom 'down'. Computes, according to the message Message, the new value of the slot "initial_code", but keeps the old value for a while. Then draws one more time the old codes, in exclusive_or, which

has the effect of erasing them, and draws the new codes instead. Then saves the new value of the slot "initial_code" in the slot itself. For aesthetic reasons, each time an old code is erased, a new one is written immediately afterwards.

redraw_display(Instance,Buffer,Code,New_code,X,Y,Espacement,Height)
Works like the local predicate draw_display/7, with the only difference that it writes two codes: instead of one, at the same position. Code and New_code. Both operations are or exclusive operations. The first one erases the code Code, previously written, the second writes the new code New_code just afterwards, at the same position.

new_code(Code,Number_of_codes,Message,^New_code)
(See the local predicates new_code, next_code and previous_code of the class font_display_window).

4.14 THE CLASS PARAMETERS_DISPLAY_MENU

4.14.1 Description

An instance of this class is not a menu in itself. It is an object which maintains a menu as an attribute. The reason why a menu, instead of a simple window, is used for only displaying parameters, is that menus know by themselves how to display the different items they have to display, and the space required for doing it, and windows do not. The reason why an instance of this class is not itself a menu, is that it is quicker to create a new menu and to show it than to modify the old one each time new parameters has to be displayed.

To ensure an easy use of instances of the class, methods are added to ensure an interface similar to the interface of a menu.

4.14.2 Instance Attributes

1. superior: is the superior window from which the top level reads inputs from the user. It must be declared as superior at the creation of the menus. Since several menus may be created, it must be saved somewhere in an attribute. See the classes as_font_display and pattern_mode_display for details.

2. menu: is the currently displayed menu.

4.14.3 Instance Methods

:set_values(Instance,List_of_values)
the list List_of_values must be of the following format: [Number1,Number2,Number3], where the numbers must be integers. Computes from the list List_of_values a list of parameters adapted to the format required by the window system for creating menus; creates a new menu, sets its size and position to be the same as the size and the position of the preceding menu, shows the new menu, kills the old one, and saves the old menu in the slot "menu".

:show(Instance)
This method, like all the following ones, are interface methods. They just call the method of the menu "menu" which has the same name.

:get_size(Instance,^Width,^Height)

:set_size(Instance,Width,Height)

:set_position(Instance,X,Y)

:get_position(Instance,^X,^Y)

:get_title(Instance,^Title)

:set_title(Instance,Title)

:kill(Instance)

4.14.4 Class Method

:create(Class,[items_list(List),superior(Superior)],~Instance)
the list List must be in the format [Number1,Number2,Number3], where the numbers must be integers. Creates a new instance of this class; saves the window Superior into the slot "superior"; computes from the list List a list of parameters adapted to the format required by the window system for creating menus; creates a menu, and saves this menu into the slot "menu".

4.14.5 Local Predicates

menu_list(Instance,[X,Y,Z],
[items_list(^Items_list),superior(~Superior)])
X, Y, and Z must be integers. For each of them, computes a string, containing the integer in decimal notation, as long as the integer is not too big or negative; in that case, uses a string composed of space characters instead. Then computes a structure adapted to the format required for creating menus, and instantiates the variable Items_list with this structure.

This structure contains the strings that the menu will display: on the first column, the strings "Width", "Height" and "Bias", on the second column, the string corresponding to the integers X, Y and Z.

get_number_string(Number,~String)

creates a string of length 4; if the integer Number is smaller than 10000 and non negative, writes it in decimal notation into the string, completing with space characters if necessary; otherwise, simply fills the string with space characters. Returns always a string of length 4, so that the minimum size of the menus created by an instance of the class will always be the same. This predicate could be simplified by using the method :writef of the class writef (see the class horz_code_display_window).

CHAPTER 5

EDITING_FACILITIES

5.1 DESIGN CONSTRAINTS

The editing facilities are the core of the font editor. They must provide to the user quick and convenient tools for editing patterns. This implies:

1. a format for recording character patterns such that the color of each dot can be accessed and modified very easily. Since this will probably not be the standard format of instances of the class character_record_with_pattern, facilities to convert the format of patterns from a format adapted to the editing to the standard format must also be provided.

2. a way of displaying the pattern being edited such that each dot of the pattern is displayed as a square of several dots wide. It must be possible to modify the pattern dot by dot, only by using the mouse; to move the pattern; to cut parts of the pattern, and to reuse old parts characters. The "reusing parts" facility is probably the most important. It requires the possibility of saving and loading patterns or parts of patterns in pattern registers, and the ability of displaying at least two patterns at the same time on the screen, say, one in grey and the other in black, which can be merged into one after their relative position has been decided.

3. Moreover, a good interface with the other parts of the editor, mainly with the top level system, is required.

5.2 DESIGN COMMITMENTS

For quick and direct access to data, a hash table seems appropriate. It has therefore be decided to store the patterns being edited in hash tables. The horizontal coordinate of each black dot is stored in the table, using the vertical coordinate as key. The coordinates of the dots are their absolute coordinates when displayed in the editing window, computed in number of squares. (The square of absolute coordinates (0,0) is the one displayed at the top left corner

of the screen). Recording the absolute coordinates of each dot that way allow quick access and modification of the color of each dot. The patterns being edited are stored in instances of the class edit_record, which provides a variety of methods for handling conversions of formats.

For displaying a pattern to be edited, a special window class has been designed: the class as_edit_pattern_window. An instance of this class is specialized in graphics. This is the only class, with the class font_graphics_window, to provide graphics methods that must be executed inside the window manager. See the class font_graphics_window for more details about this problem.

Another class, the class edit_pattern_window, has been designed for providing interfacing methods with the top level system, that the class as_edit_pattern_window does not provide. The class edit_pattern_window inherits the class as_edit_pattern.

5.3 THE CLASS EDIT_RECORD

5.3.1 Description

An instance of this class is in charge of keeping, in a hash table, with direct and easy access, a pattern that is being edited. It stores only the absolute coordinates of the black dots of the pattern separately. The horizontal coordinate is stored in the hash table with key the vertical coordinate. It provides methods for modifying the pattern dot by dot.

It is also in charge of converting, in the standard format of instances of the class char_record_with_pattern, parts or the totality of the pattern it maintains in a hash table. The whole pattern format is required when the pattern has to be saved in a register; only a part of the pattern is converted when the user uses the "CUT" operation.

Moreover, it is in charge of keeping the information about the dimensions of the box which is displayed in the window used for editing patterns. The dimensions of the box can be modified at any time by the user, and correspond to the size and the bias the user wishes to give to the final character. This information is used when converting the pattern being edited into the definitive format in which it will be sent to the character mode. In the definitive format, only the part of the pattern which lies inside the box is saved; its width, height and bias are set to be the width, the height and the bias of the box.

It also provides a method for converting old patterns stored in the char_record_with_pattern format into the hash table format, and a method for translating the pattern. This last method can be used for converting absolute coordinates to relative coordinates, relative coordinates to absolute coordinates, or simply for moving the pattern on the screen. It also provides method for modifying the size and the position of the box.

The window used for editing patterns stores all the information it needs for drawing its display into two instances of this class, one of

which is displayed in solid black, the other in grey. It is therefore straightforward to exchange the pattern displayed in grey and the pattern displayed in black in this window; what is less straightforward and which is provided as an instance method of this class, is the function which adds another pattern received from outside to the pattern stored into the hash table.

5.3.2 Instance Attribute

An instance of this class has 11 attributes.

1. grid_size: This parameter has already been introduced. It is the size of the squares used for representing the dots of the patterns displayed in the window used for editing them. It is used only for converting the size of the editing window from number of dots to number of squares. It must be suppressed, since the grid size has nothing to do with patterns. The size of the window used for editing patterns must be directly sent to this instance computed in number of squares. In the present version, must be initialized just after the creation of an instance.

2. window_width: is the width of the window used for editing patterns. See the instance attribute "grid_size". Must be initialized just after the creation of an instance of this class.

3. window_height: is the height of the window used for editing patterns. See the instance attribute "grid_size". Must be initialized just after the creation of an instance of this class.

4. box_x: is the absolute horizontal coordinate of the upper left corner of the box, as displayed in the window used for editing patterns. The coordinate is computed in number of squares. Must be initialized just after the creation of an instance of this class.

5. box_y: is the absolute vertical coordinate of the upper left corner of the box, as displayed in the window used for editing patterns. The coordinate is computed in number of squares. Must be initialized just after the creation of an instance of this class.

6. box_width: is the width of the box, as displayed in the window used for editing patterns. It is computed in number of squares. Its default value is 13, the width of the characters of the default font "font_13".

7. box_height: is the height of the box, as displayed in the window used for editing patterns. It is computed in number of squares. Its default value is 19, the height of the characters of the default font "font_13".

8. box_base: is the base of the box, which is the distance between the upper line and the middle line of the box, as displayed in the window used for editing patterns. The relation between the base of the box and the bias of the pattern being edited is:
   Bias = box_height - box_base
   It is computed in number of squares. Its default value is 3, the bias of the characters of the default font "font_13".

9. record: this slot is used for storing instances of the class char_record_with_pattern, during format conversion operations.

10. line_list: is an instance of the class list. It contains the list of the lines, in absolute coordinates, which contains at least one black dot.

11. edit_pattern: is an instance of the class hash_index. It contains the coordinates of the black dots of the pattern being edited, The vertical coordinates are used as keys, the horizontal coordinates as entries.

5.3.3 Instance Methods

:set_window_size(Instance,Window_width,Window_height)

:get_grid_size(Instance,^Grid_size)

:set_grid_size(Instance,Grid_size)

:set_box_size(Instance,Box_width,Box_height,Box_base)
modifies the values of Box_width, Box_height and Box_base so that the box fits into the window used for editing patterns, and so that the base lies in the interval [0,Box_height] and saves the resulting values into the corresponding slots.

:get_box_size(Instance,^Box_x,^Box_y,^Box_width,
^Box_height,^Box_base)

:set_box_size(Instance,Box_x,Box_y,Box_width,Box_height,Box_base)
modifies the values of Box_x, Box_y, Box_width, Box_height and Box_base so that the box fits into the window used for editing patterns, and so that the base lies in the interval [0,Box_height], and saves the resulting values into the corresponding slots.

:center_box(Instance)
checks if the dimensions of the box fits the size of the window, in the case the user had changed the grid size and therefore the size of the window in number of squares; then computes the values to give to the slots "box_x" and "box_y" so that the box is centered in the middle of the window used for editing patterns; then translates the pattern stored is the hash table the same way the box has been moved, so that

the relative position of the pattern and the box remains the same.

:move_box(Instance,Vector_x,Vector_y)
translates the box using (Vector_x,Vector_y) as the vector associated to the translation. Stops the move before the box goes out of the window used for editing patterns.

:move_pattern(Instance,Vector_x,Vector_y)
translates the pattern stored in the hash table "edit_pattern" using (Vector_x,Vector_y) as the vector associated to the translation. Translating outside the window is allowed. The dots lying outside the window are simply not displayed.

:get_line_list(Instance,^Line_list_contents)

:get_whole_size_and_position(Instance,^Width,^Height,
^Char_x,^Char_y)
If there is no pattern recorded in the hash table "edit_pattern", fails. Otherwise, computes the size and the position of the smallest rectangle that contains the pattern stored into the table "edit_pattern".

:get_record(Instance,~Record)
If the table "edit_pattern" is empty, instantiates the variable Record to the atom 'nil'. Otherwise, saves the part of the pattern, stored in the table, which lies inside the box into a new instance of the class char_record_with_pattern. Saves also in this record the width, the height and the bias of the box. It is used for sending a character pattern to the character mode.

:get_whole_record(Instance,~Record)
If the table is empty, instantiates the variable Record to the atom 'nil'. Otherwise, saves the whole pattern stored in the hash table into a new instance of the class char_record_with_pattern. Computes the exterior size of the pattern, and saves it into the new record. Saves also into the record the position of the pattern. It is the format used for saving a pattern into a register.

:get_cut_record(Instance,X0,Y0,X1,Y1,~Record)
(X0,Y0) and (X1,Y1) must be the absolute coordinates of two opposite vertices of a rectangle. If the table "edit_pattern" is empty, instantiates the variable Record to the atom 'nil'. Otherwise, saves the part of the pattern which lies inside the rectangle defined by the absolute coordinates of a couple of opposite vertices into a new instance of the class char_record_with_pattern. Saves also in the new record the width and the height of the rectangle, as well as the coordinates of its upper left vertex. This format is used for saving a part of a pattern into a register.

:set_record(Instance,Record)
Must be paired with the method :get_record. Clears the contents of the slots "line_list" and "edit_pattern" and centers the box; saves the record Record into the slot

"record"; if Record is equal to the atom 'nil', stops; otherwise, sets the size of the box to be equal to the size of the record (the bias of the record is translated into the base of the box using the equation: base = height - bias); loads the pattern of the record into the table "edit_pattern", the vertical coordinates of the lines which contain at least one black dot of the pattern being saved into the list "line_list". Loads the pattern in such a way that the position of its top left corner is (0,0). Then moves the pattern so that its top left corner coincides with the top left corner of the box. Thus the pattern lies exactly inside the box, which has the size of the pattern.

:set_whole_record(Instance,Record)

Must be paired with the method :get_whole_record. Clears the contents of the slots "line_list" and "edit_pattern"; saves the record Record into the slot "record"; if Record is equal to the atom 'nil', stops; otherwise,loads the pattern of the record into the table "edit_pattern", the vertical coordinates of the lines which contain at least one black dot of the pattern being saved into the list "line_list". Loads the pattern in such a way that the position of its top left corner is (0,0). Then moves the pattern so that its top left corner has the position it had on the screen before it was saved in the record Record by using either the method :get_whole_record or the method :get_cut_record.

It must be noted that only instances of this class are authorized to create instances of the class char_record_with_pattern (at the exception of the instances of the class edited_font, which are only allowed to generate new copies of already existing patterns). Only three methods can generate new instances of this class: the methods :get_record, :get_whole_record and :get_cut_record. The first one generates a definitive format, to be sent to the character mode; this format is understood by the method :set_record, which is used when an old pattern is going to be modified. The last two ones generates a format suitable for saving the record into pattern registers; this format is understood by the method :set_whole_record.

:get_dot_color(Instance,X,Y,^Color)

(X,Y) are the absolute coordinates of a dot on the screen of the editing window, in number of squares. Color is instantiated to the atom 'black' if the the dot (X,Y) is recorded in the hash table "edit_pattern", to the atom 'white' otherwise.

:set_dot_color(Instance,X,Y,Color)

(X,Y) are the absolute coordinates of a dot on the screen of the editing window, in number of squares. If the color of the dot of coordinates (X,Y) is already Color, does nothing; otherwise, if the color Color is 'black', records the dot (X,Y); if the color Color is 'white', removes the dot (X,Y).

:empty(Instance)

Succeeds if the table "edit_pattern" is empty, fails

otherwise.

:get_line(Instance,^List,Y)
Instantiates the variable List with a (stack vector) list containing the horizontal coordinates of the black dots, of vertical coordinate Y, of the pattern stored in the table "edit_pattern". The list may be empty.

:add_pattern(Instance,Record)
Record comes from another instance of this class, generated by a call to the method :get_whole_record. If Record is equal to the atom 'nil', does nothing. Otherwise, if the table "edit_pattern" is empty, calls the method :set_whole_record for loading the record Record into the table. Otherwise, moves the pattern recorded in the table to the position (0,0), saves the record Record into the slot "record", adds the pattern of this record to the table, and moves back the contents of the table to its initial position.
No new local predicate was introduced for implementing this method. On the other hand, the display of the result of this operation cannot be optimized by using this implementation, because it does not provide any way to remember which squares have been added.

:clear(Instance)
centers the box by calling the method :center_box; clears the table "edit_pattern" and the list "line_list".

5.3.4 Class Methods

:create(Class,~Instance)
Identical to :new(Class,Instance).

:create(Class,Record,~Instance)
creates a new instance of this class, and loads the pattern of the record Record in it, using the method :set_record.

5.3.5 Local Predicates

get_record(Instance,~Record)
(See the instance method :get_record).

load_record(Instance,Record)
Record must have been saved already in the slot "record". The second argument of this predicate is therefore not very useful, and can be replaced with profit by the line of code:
Record = Instance!record
Saves into the record Record the width, the height and the bias of the box; moves the pattern stored in the table "edit_pattern" in such a way that the dot displayed in the upper left corner of the box is translated to the upper left

corner of the window screen, which is of coordinates (0,0). Then saves in a double_bytes string the part of the pattern recorded in the table which lies inside the rectangle of top left vertex (0,0), and of size the width and the height of the box, and saves this string in the record. The string is in the format required for the patterns stored in char_record_with_pattern instances. Finally moves back the pattern stored in the table to its original position.

get_whole_record(Instance,~Record)
(See the instance method :get_whole_record).

load_whole_record(Instance,Record)
Record must have been saved in the slot "record". The second argument of this predicate is therefore not very useful, and can be replaced with profit by the line of code:
Record = Instance!record
Computes the size and the position of the smaller rectangle that contains the pattern stored in the table "edit_pattern". Saves these size and position in the record slots, moves the pattern in such a way that the position of the top left corner of the pattern goes in (0,0), saves it in a double_bytes string and saves the string in the record. The string is in the format required for the patterns stored in char_record_with_pattern instances. Finally moves back the pattern stored in the table to its original position.

get_cut_record(Instance,X,Y,Width,Height,~Record)
(See the instance method :get_cut_record).

load_cut_record(Instance,X,Y,Width,Height,Record)
Record must have been saved in the slot "record". As before, the last argument of this predicate is not really useful.
Saves the four parameters (X,Y,Width,Height) in the record slots, moves the pattern in such a way that the dot of coordinates (X,Y) is translated to the dot of coordinates (0,0), saves in a double_bytes string the part of the pattern, recorded in the table "edit_pattern", which lies inside the rectangle of position (0,0) and of size (Width,Height), and saves this string in the record. Then moves back the pattern stored in the table to its original position.

adjust_box_size(Instance,Width,Height,Base)
Checks the proposed size and base of the box, modifies them if necessary in such a way that the box is kept inside the editing window, and saves these values into the corresponding slots.

center_box(Instance)

load_edit_pattern(Instance)
saves in the hash table "edit_pattern" the contents of the record "record", and modifies the list "line_list" consistently. For details about the parameter raster_width, see the class char_record_with_pattern.

put_string_into_table(Instance,String,Width,Raster_width, Height,H)

The horizontal coordinates of the dots of vertical coordinates H is stored in the substring of the string String of position H * Raster_width and of length Raster_width. Raster_width is not necessarily the smallest number of double bytes required for storing a line of dots of width Width. It may be larger than the minimum value. In that case, the first double bytes of the substring does not contain any relevant information and can be thrown away.
It is what this predicate does: it takes from the string String only the interesting part of it. The position of this part is (H + 1) * Raster_width - Min_raster_width, and its size is Min_raster_width, where Min_raster_width is the smallest number of double bytes required for storing a line of dots of width Width.
Then it saves the contents of this substring in the edit format, increases H by one, and calls itself recursively. It stops when H becomes equal to Height.

put_one_line(Object,Substring,Y,Width,Raster_width,N)

This time, Raster_width must be equal to the smallest number of double bytes required for storing a line of dots of width Width. It must also be equal to the length of the string Substring.

Each double byte is an integer smaller than (2 power 16). In base 2, this integer can be uniquely decomposed in the following form:
X = X0 * (2^15) + ... + X15 * (2^0)
where X0, X1, ..., X15 are equal either to 0 or 1.

For saving a line of width 16 exactly, it is really easy to use a double byte. The only thing to do for computing it is to specify the values of the quantities X0, X1, ... X15 which appears in its decomposition in base 2. And there is a natural way to do so: if the dot of position i in the line is black, Xi is set to 1; otherwise Xi is set to 0.

For saving a line of width 32, 48, 64, or any multiple of 16, it is not more difficult really: the color of the 16 first dots of the line are saved in a double byte as before, and for the rest of the line, this procedure is called recursively.

But what happens if the width of the line is not a multiple of 16 ? A new line is made from it, by adding white dots to its left, until its width becomes a multiple of 16, and it is this new line which is stored in double bytes.

This predicate is doing the inverse operation: it takes from the string Substring the double byte which lies at the position N,and translates the information it contains into the horizontal coordinates of the black dots of vertical coordinates Y, and stores these coordinates into the hash table.

In the case of the first double byte, it needs to know the number of white dots that have been added at the left of the line when it has been stored in the string. This number is:
A = 16 * Raster_width - Width.
Thus, in the decomposition in base 2 of the first double byte, only the values of XA, ..., X15 correspond to dots of the line. The horizontal coordinate of the dot of the line corresponding to XA is 0.

In the case of the other double bytes, all the Xi correspond to dots of the line. The only important parameter in that case is the horizontal coordinate of the dot of the line corresponding to X0. It easy to see that this coordinate is equal to:
16 * N - ((16 * Raster_width) - Width).

put_one_element(Instance,Double_byte,Y,X,Pointer)
If, in the decomposition of Double_byte in base 2 the quantity 2 power (15 - Pointer) appears, removes it, stores the dot (X,Y) in the table "edit_pattern", and, if necessary, the vertical coordinate Y in the list "line_list"; otherwise does nothing. Then always increases X and Pointer by 1, and calls itself recursively. Stops when Double_byte becomes equal to 0.

save_edit_pattern(Instance)
Supposes that the slot "record" contains an instance of the class char_record_with_pattern, and that the size of this record has been recorded in its slots. Asks to this record its size, computes the number of double bytes necessary to store a line of its pattern. Creates a double bytes string of length this number times the height of the record. Then stores in this string the part of the pattern saved in the table "edit_pattern" which begins at the position (0,0) and is of the same size as the record. Then saves this string in the record "record".

put_table_into_string(Instance,String,Width,Raster_width,
Height,Line_list)
Removes from the list Line_list the vertical coordinate H of the next line. If this coordinate is negative, or larger than or equal to Height, does nothing. Otherwise, computes the number of white dots it has to add to the left of this line (see the local predicate put_one_line for details), creates a string of length Raster_width to store this line in, stores this line in this new string, and copies the contents of this new string in the string String, at the position corresponding to the line, that is H * Raster_height. Then calls itself recursively. Stops when the list Line_list is empty.

put_list_into_buffer(Dots_list,Substring,Width,Raster_width,
Lost_space)
Lost_space must be instantiated to the number of white dots that must be added at the left end of the line (see the local predicate put_one_line). Removes the horizontal coordinate X of the next dot from the list Dots_list; if it is negative,

or larger than or equal to Width, does nothing. Otherwise, shifts X by the number of white dots that must be added on the left, computes in which double byte string of the string Substring X must be recorded, and then computes the power of 2 that must be added to this double byte for X to be saved. If X is divided by 16, one can write:
X = 16 * Quotient + Remainder
Quotient is the position of this double byte in Substring, and remainder the position of the black dot of coordinate X in this double byte. Therefore, the power of 2 that must be added to this double byte is (2 power (15 - Remainder)).

get_dot_color(Instance,X,Y,^Color)

set_dot_color(Instance,X,Y,Color)

add_list(Instance,X,Y)
This method is supposed to be used just after the dot (X,Y) has been stored in the table "edit_pattern". Checks if the contents of the line of vertical coordinate Y are composed of the dot (X,Y) only; in that case, adds Y to the list "line_list". Otherwise, does nothing.

remove_list(Instance,X,Y)
This method is supposed to be used just after the dot (X,Y) has been removed from the table "edit_pattern". Checks if the line of vertical coordinate Y is empty or not; if it is empty, removes Y from the list "line_list". Otherwise, does nothing.

raster_width(Width,^Raster_width)

check_dimensions(Instance,Input_width,Input_height,Input_base,
^Box_width,^Box_height,^Box_base)
If Input_width is negative, instantiates Box_width to 0; if Input_width is larger than the width of the editing window, in number of squares, instantiates Box_width to the width of this window; otherwise instantiates Box_width to the value of Input_width. Does the same for Box_height and Box_base. The limits for Box_base are 0 and the height of the window.

check_dimensions(Instance,Input_x,Input_y,Input_width,Input_height,
Input_base,^Box_x,^Box_y,^Box_width,
^Box_height,^Box_base)
Computes Box_width, Box_height and Box_base by calling the local predicate check_dimensions/7; then if Input_x is negative, instantiates Box_x to 0; if Input_x + Box_width is greater than the width of the window, in number of squares, then instantiates Box_x to the difference between the window width and the value of Box_width; does the same for Box_y.

move_pattern(Instance,Vector_x,Vector_y)
Creates a new hash table, and translates the pattern recorded in the table "edit_pattern" and stores the result in this new table. The vector of the translation is (Vector_x,Vector_y).

move_pattern(Instance,Table,Line_list,^New_line_list,
Vector_x,Vector_y)
Removes the vertical coordinate of the next line from the list Line_List; translates this coordinate of Vector_y, and if the new vertical coordinate is negative, increases it by (2 power 16). The reason of doing so is that the vertical coordinate of the lines are used as keys of hash indexes, and that hash index do not accept negative numbers as keys. For avoiding this problem, the keys are computed modulo a large number: 2 power 16.
Then translates the dots of the next line by Vector_x and stores the result into the table Table, using for key the translated vertical coordinate. Then calls itself recursively. Stops when the list Line_list is empty.

move_line(Instance,Table,Y,New_y,Dots_list,Vector_x)
For all the horizontal coordinates X appearing in the list Dots_list, removes the dot (X,Y) from the table "edit_pattern", which is perfectly useless, since this table will never be used again, and adds the dot (X + Vector_x,New_y) to the table Table.

move_box(Instance,Vector_x,Vector_y)
translates the box of (Vector_x,Vector_y) as long as the box stays inside the window used for editing patterns. If the vector (Vector_x,Vector_y) is too large, it is cut so that the translation keeps the box just inside the window.

compute_size_and_position(Instance,Line_list,
Min_x,Min_y,Max_x,Max_y)
removes the vertical coordinate of the next line from the list Line_list, and calls itself recursively. If this coordinate is larger than (2 power 15), it is considered that this coordinate is indeed negative, and that what was recorded in the list is a positive integer, equivalent modulo (2 power 16). Thus, in that case, this coordinate is decreased by (2 power 16). With this value, and the values computed recursively, determines the values of Min_y and Max_y. Then computes the minimum and the maximum values of the horizontal coordinates of the black dots of the next line, and determines the values of Min_x and Max_x using the values computed recursively.

compute_extremums(List_of_integers,^Smallest,^Largest)

5.4 THE CLASS AS_EDIT_PATTERN_WINDOW

5.4.1 Description

This class inherits only the basic window class: font_inferior_window. It is the most important window class used by the font editor.

An instance of this class is in charge to provide to the user basic facilities to edit a pattern. First of all, it provides a grid, which divides the screen of the window in regular squares. Each of these squares can be accessed by the user, who can change their color from white to black or from black to white by simply clicking the mouse on them. The pattern composed of these blackened squares is called the black pattern. This pattern can be saved at any moment in a pattern register or sent to the character mode for creating a new character.

An instance of this class provides also one register for storing a pattern. The contents of this register are displayed in the window itself, as if it were the black pattern, with the only difference that the dots are displayed in grey instead of being displayed in black. The contents of this register form what is called the grey pattern. The user can make the difference de visu between the black pattern and the grey pattern.

It displays also a box, which mission is to help the user in setting the final size and bias of the character pattern, and provides methods for modifying the size and the position of the box.

All the structures displayed on the screen: the grid, the black and the grey patterns and the box, are added in the display rather than copied. This creates a screen image reasonably easy to understand, and has the strong advantage of allowing a modular implementation of the display methods. Moreover, the problem of distinguishing between dots of the grey pattern only, dots of the black pattern only, and dots which are both of the grey pattern and of the black pattern is simply solved by displaying the dots of the grey pattern in light grey. Wherever a black dot is added to a grey dot, the result will appear in dark grey.

To speed up the display, two patterns, corresponding respectively to a black square and a grey square are kept in attributes, in the double bytes string format adapted to the interface of the graphics methods provided by the window system. They are computed only at the creation of the instance, and each time the size of the grid is modified by the user.

The choice of the pattern used for displaying grey squares was not easy. First of all, a pattern using one third of black dots and two third of whites was tried, but appeared to be difficult to distinguish from the inverse pattern. Then a regular pattern composed of one fourth of black dots and three fourths of whites has been tried, but turned to be unusable because of a important flickering of the screen. The present version is also a pattern composed of one fourth of black dots and three fourths of whites, which avoids flickering by repeating twice each line of pattern.

The black pattern and the grey pattern are stored each in a different instance of the class edit_record, which allows easy

modifications of the patterns. Even if the grey pattern cannot be directly modified, it is stored in exactly the same format as the black pattern, mainly because the operation of exchanging these two patterns is a very common one, and must be kept as simple as possible. However, the size and position of the box are stored only in the black record, and must be transmitted from one record to the other when the black and the grey patterns are exchanged.

At last, a method has been added to allow the user to draw a pattern simply by moving the mouse on the screen. The implementation of this method is the only one ever concerned with efficiency in the whole system. It gives an idea of the performance that can be expected from the font editor and the PSI machine.

5.4.2 Instance Attributes

1. grid_size: well known parameter; contains the size, in number of dots, of square of the grid. Its value can be changed by the user. The admissible values run from 5 to 25.

2. draw_area: this slot is used by the local predicates in charge of drawing the patterns. They keep in it a bitmap area containing the pattern of a black or a grey square. When not used, has the value 'nil'.

3. operation: is a fixed attribute, initialized at the value exclusive_or. The graphics instance methods of this class all use the exclusive_or operation.

4. grey_string: contains a double byte string in which the pattern of a grey square of size "grid_size" is recorded. Its contents are modified each time the grid size is modified.

5. black_string: contains a double byte string in which the pattern of a black square of size "grid_size" is recorded. Its contents are modified each time the grid size is modified.

6. grid_string: contains a double byte string in which a pattern corresponding to one line of the grid of the screen is stored. This pattern is composed of regular intervals of length grid size, such that, in an interval, the only black dots are the first and the last. This pattern is used for drawing the grid of the window. It is modified each time the grid size is modified. Since it is only used once, it may not be necessary to save it as an attribute.

7. empty_string: contains a double bytes string in which a white line of the screen is stored. For drawing the grid of the window, a bitmap area of the size of the window screen is allocated, the grid is drawn in this area, and then the area is transferred into the bitmap memory displayed in the screen of the window. Since the window system does not usually clear the areas when allocating them, it was necessary to clear the lines

of the area, not filled by the pattern stored in the slot "grid_string". It is what the pattern contained in this slot is used for. A better algorithm would be to clear the area first, using a window system method, and to use only the "grid_string" when required.

8. black_record: saves the instance of the class edit_record in which the black pattern and the size and dimension of the box are recorded.

9. grey_record: saves the instance of the class edit_record in which the grey pattern is recorded.

5.4.3 Instance Methods

:set_grid_size(Instance,Grid_size)
saves the value of the variable Grid_size into the slot "grid_size"; the value must have been checked before; computes the patterns corresponding to a black and a grey squares of size Grid_size, and stores them into the slots "grey_string" and "black_string"; computes the patterns used for drawing the grid and stores them in the corresponding slots: "grid_string" and "empty_string"; sets the inside units of the window to (grid_size,grid_size), so that the position of the mouse is directly read from the window system in number of squares instead of being read in number of dots; then initializes the two instances of the class edit_record saved in the slots "black_record" and "grey_record", by giving them the size of the window, the value of the grid size and by asking them to set the box position in the middle of the screen.
As mentioned before, (see the class edit_record), the instances of the class edit_record are not concerned in the real size of the window, neither with the value of the grid size, but rather in the size in number of squares of the window. This should be modified.

:get_grid_size(Instance,^Grid_size)

:get_operation(Instance,^Operation)

:set_operation(Instance,Operation)

:center_box(Instance)
Centers the box in both the "black_record" and the "grey_record".

:draw_display(Instance)
clears the window, draws the grid, draws the black record, draws the grey record and draws the box. All these operations are or exclusive. They all use methods that must be executed inside the window manager, except the method which draws the box.

:draw_char_box(Instance)
Draws the box, using an exclusive or operation.

:draw_record(Instance,Message)
Message can be one of the three following atoms: 'black_record', 'grey_record' or 'all'. Calls, through the window manager, the method :do_draw_record for drawing either the black record, the grey record, or both.

:do_draw_record(Instance,Message)
Message can be one of the three following atoms: 'black_record', 'grey_record' or 'all'. Must be executed inside the window manager. Draws either the black record, the grey record, or both, according to the message Message.

:blacken_dot(Instance,X,Y)
(X,Y) corresponds to the position of the mouse on the screen, computed in number of squares, and therefore to the absolute coordinates of a dot of the black pattern. If this dot is already black, does nothing; otherwise, records this dot as black in the record "black_record" and inverses the color of the corresponding square on the screen.

:whiten_dot(Instance,X,Y)
similar to the method :blacken_dot.

:inverse_dot(Instance,X,Y)
similar to the methods :blacken_dot and :whiten_dot.

:inverse_square(Instance,X,Y)
(X,Y) must be the coordinates of a square in the screen. Calls, through the window manager, the method :do_inverse_square for inverting the color of the square.

:do_inverse_square(Instance,X,Y)
Must be executed inside the window manager. Inverses the color of the square of coordinates (X,Y) on the window screen.

:draw_curve(Instance,Color)
allows the user to draw a black pattern only by moving the mouse on the screen. For avoiding too much overhead, asks to the record "black_record" the address of the hash table in which it stores the black pattern, for writing itself directly in it.

5.4.4 Class Method

after:create(Class,Parameters_list,~Instance)
erases the indesirable cursor; sets the grid size to the default value 20, by calling the instance method :set_grid_size which furthermore initializes the various slots of the instance Instance. But the instance of the class pattern_mode_display, which creates this instance, also calls the method :set_grid_size. Thus the job is done twice,

without any reason. This must be modified. Probably, to eliminate the call to the method :set_grid_size in this demon will be the solution. But it must be checked that this solution would not create another problem somewhere else.

5.4.5 Local Predicate

draw_display(Instance)
draws the grid, the black record, the grey record and the box.

draw_grid(Instance)
This local predicate is executed inside the window manager. Therefore the window system methods used in it must be manager methods, not user methods. The manager method corresponding to the user method :get_size(Window) is :do_get_size(Window).
Asks the size of the window; allocates an area of the same height than the window, and of width the first multiple of 16 greater than or equal to the width of the window. The reason of that manipulation is that it is possible to write in a bitmap area only by using double bytes buffers or strings. Writes the grid into the area, leaving unchanged the columns at the left of the area which outnumber the columns of the window; transfers the contents of the area in the window screen memory, and deallocates the area.

draw_grid(Instance,Area,Area_width,Grid_size,Height,H)
If the remainder of the division of H by Grid_size is equal to 0 or to (Grid_size - 1), draws, in the area Area, the line contained in the string "grid_string"; otherwise, draws the line of white dots contained in the string "empty_string"; then increases H by 1, and calls itself recursively. Stops when H is equal to Height.

draw_char_box(Object)
asks to the record "black_record" the size and position of the box; translates them from square coordinates to dots coordinates, and draws the rectangular frame of the box. Then draws the base line of the box, except when this line corresponds to a part of the frame. This exception cannot be avoided since the lines are drawn using an exclusive or operation.

draw_record(Instance,Record_name)
gets the pattern of the record of name Record_name, which is either the black or the grey record, and draws the pattern in the corresponding color.

draw_record(Instance,Record_name,Record_string)
allocates an area of height the grid size, of width the first multiple of 16 greater than or equal to the grid size; draws the pattern Record_string in it, which fills the area either in black or in grey; saves the area in the slot "draw_area"; draws the record stored in the slot of name Record_name;

deallocates the area Area and sets back the slot "draw_area" to 'nil'.

draw_pattern(Instance,Record_name,Line_list)
removes from the list Line_list the next vertical coordinate of a line of the pattern, recorded in the slot Record_name, from the list Line_list; draws this line on the screen , and calls itself recursively. Stops when the list Line_list is empty.

draw_line(Instance,Record_name,Dots_list,Y)
removes from the list Dots_list the next horizontal coordinate of a dot of vertical coordinate Y, of the pattern recorded in the slot Record_name; checks if this dot corresponds to a square inside the window; if it is the case, draws the dot on the screen; otherwise, does nothing. Then calls itself recursively. Stops when the list Dots_list is empty.

draw_element(Instance,Record_name,X,Y)
transfers into the window screen the black or the grey square stored in the area "draw_area" at the position (X,Y), computed in number of squares. Since the area is filled by a black or a grey pattern, any square in this area of size grid size can be used to transfer its contents in the window screen and to obtain the desired result. The variable Record_name is not used.

compute_grid_string(Instance,~Grid_string,~Empty_string)
computes the smallest number of double bytes necessary to cover a full line of the window screen; creates two strings of this size, Grid_string and Empty_string, and loads in the string Grid_string the dots used for displaying the grid squares on the screen.

load_grid_string(Grid_string,Grid_size,Lost_space,Width,N)
Width corresponds to the width of the window; Lost_space to the difference between the width of the Grid_string (which is its length times 16, and which is expected to be the smallest multiple of 16 greater than or equal to Width). If the remainder of the division of N by Grid_size is equal to 0 or to Grid_size - 1, sets to 1 in the string Grid_string the bit at the position N + Lost_space; then increases N to the next integer which is congruent to 0 or to Grid_size - 1 modulo Grid_size, and calls itself recursively. Stops when N is greater than or equal to Width.

compute_grid_point(.....)
This local predicate has only an historical interest, and is no longer used. Can be suppressed.

adapt_to_double_bytes_format(Width,^Adapted_width)
Instantiates Adapted_width to the smaller multiple of 16 greater than or equal to the value of the variable Width.

inverse_square(Instance,X,Y)
draws a black square at the position (X,Y), computed in square

coordinates, on the window screen, by using an exclusive or operation; see the local predicates draw_record/3 and draw_element. As these methods, must be executed inside the window manager. Does not check if the dot (X,Y) is inside the window or not. Presently, it is only used by methods receiving directly the parameters (X,Y) as positions of the mouse on the screen. In that case, the check is implicit.

outside_display(Instance,Record_name,X,Y)
succeeds if the square of coordinates (X,Y) lies outside the window screen; fails otherwise. Must be executed inside the window manager. The variable Record_name is not used.

compute_grey_string(Instance,~Grey_string)
creates a buffer, which is a double byte string just long enough to store a line of width grid size; this buffer will be used for storing one line of the grey square; creates the string Grey_string, of length the length of the buffer times the height of the grey square, that is, the grid size. Then computes a double byte integer which corresponds to a regular pattern of one black dot every four dots. Shifts this double byte of one bit and of three bits, to form the two basic double bytes that will be used as parts of the grey square. Then loads the pattern into the string Grey_string, line by line, repeating, for two consecutive lines, the first double byte, and then for two consecutive lines the second double byte.

compute_black_string(Instance,~Black_string)
creates a buffer, which will be a double byte string just long enough to store a line of width grid size; this buffer will be used for storing one line of the black square; creates the string Black_string, of length the length of the buffer times the height of the black square, that is, the grid size. Then computes the double byte integer which corresponds to a line of black dots. Then loads the pattern into the string Black_string, line by line, repeating this double byte in each line.

load_pattern(String,Buffer,Raster_width,Double_bytes_list,
Step,Pointer)
Buffer must be a double byte string of length Raster_width, and String a double byte string of length a multiple of Raster_width. Removes the next double byte from the list Double_bytes_list; fills the string Buffer with this double byte, copies the string Buffer into the string String at the positions Pointer, and all the positions obtained by increasing Pointer by a multiple of Step * Raster_width, until no more space is available. Then increases Pointer by 1, modulo Step, and calls itself recursively. Stops when the list Double_bytes_list is empty.

load_buffer(Buffer,X,Buffer_length,N)
puts the double byte X in the string Buffer at the position N, increases N by 1, and calls itself recursively. Stops when N is equal to Buffer_length.

load_string(String,Buffer,Step,Buffer_length,Height,H)

String must be a string of length Buffer_length times Height. Copies the string Buffer into the string String at the position H * Buffer_length, increases H by Step, and calls itself recursively. Stops when H is greater than or equal to Height.

draw_curve(Instance,Table,Color)

allocates an area corresponding to a square and loads in it the pattern "black_string", in much the same way that it is done in the local predicates inverse_square or draw_record. Since this predicate is not itself executed inside the window manager, it must send messages to the window manager for executing manager methods. Waits for an input. If the input is a left click from the mouse, puts the font editor in a mode allowing the user to draw a pattern on the screen only by moving the mouse. Otherwise does nothing. At last always deallocates the area.

draw_curve(Instance,Area,Table,Color)

The table Table must be the hash table used by the record "black_record" for storing the black pattern. Color can be either 'black' or 'white'. Enters a (repeat - fail) loop. Reads the input from the screen without waiting. If the input is a left click from the mouse, or if the mouse is outside the window, assures the consistency between the list of the lines used by the black pattern and the contents of the slot "line_list" of the record "black_record", and stops. If the input is a middle click of the mouse, changes the color, from black to white or from white to black; if the input is a right click, and if the color is black, does nothing and waits for another input; when this input arrives, fails, and therefore resumes from the repeat instruction; if the color is white, then calls the local predicate erase.

Otherwise, if the square, on which the mouse is, has the color Color, does nothing; otherwise, modifies the record of the square in the table Table, by inserting it if the color is black, or deleting it if the color is white, inverses the corresponding square on the screen, and fails back to the repeat instruction.

erase(Instance,Area,Table)

Is very similar to the local predicate draw_curve/4, when the argument Color is equal to 'white'. There are only two differences. When a right mouse click is read, this predicate calls the local predicate draw_curve/4, with the color 'white', and not itself; moreover, instead of only setting to white the square pointed by the mouse, it sets also to white all the eight adjacent squares.

The meanings of the different clicks are not at all obvious or natural for the user, and must be modified. The clicks must have the same meaning, independently of the color used; the facility to stop for a while is as useful when the color is white as when the color is black. It will be of great help for the user to show which color is currently selected, for example by changing the shape of the mouse.

reset_line_list(Instance,Table)
gets the list stored in the slot "line_list" of the black record; checks, for all the lines of squares displayed in the window, if there is a black dot of the black pattern in it or not, and modifies the list "line_list" depending on the result of the check.

reset_line_list(Instance,Table,Line_list,Height,H)
if the line of vertical coordinate H does not contain a black dot of the pattern stored in the table Table, removes H from the list Line_list if necessary; otherwise, adds H to the list Line_list if necessary, increases H by 1 and calls itself recursively. Stops when H is equal to Height.

5.5 THE CLASS EDIT_PATTERN_WINDOW

5.5.1 Description

This class inherits the class as_edit_pattern_window, and has for only responsibility to translate the commands coming from the top level into inherited methods. Since all the basic methods exist already, it has just to put the pieces together.

5.5.2 Instance Attributes

1. brush_color: this attribute can take three different values: the atoms 'black', 'white' and 'inverse'. Depending on the contents of this slot, a left click of the mouse on a square of this window will set its color to black or white, or will inverse its color. The user can change the value of this attribute by selecting the corresponding entry of the brush menu, or by clicking double left with the mouse (see the class pattern_mode).

2. brush_shape: this attribute can store three different values: the atoms 'dot', 'line' and 'square'. Depending on the contents of this slot, a left click of the mouse on a square of this window will affect either only this square, or this square and the two adjacent ones on the left and on the right, or this square and all the eight adjacent ones. The effect depends on the color selected, but is the same for all the squares concerned. The user can change the value of this attribute by selecting the corresponding entry of the brush menu. (see the class pattern_mode).

5.5.3 Instance Methods

:set_black_record(Instance,Record)
Record must be an instance of the class char_record_with_pattern, previously recorded in an instance of the class edited_font. Erases on the screen the box and the black pattern, simply by drawing them one more time using an exclusive or operation; saves the record Record into the black record, which is an instance of the class edit_record; the pattern of the record Record is then stored in the black record. Then draws this new black pattern, and draws the box again. The record Record contains implicit information about the size and the position of the box, which therefore may have changed.

:set_grey_record(Instance,Record)
Record must be an instance of the class char_record_with_pattern, previously recorded in an instance of the class edited_font. Erases on the screen the grey pattern, simply by drawing it one more time using an exclusive or operation; saves the record Record into the grey record, which is an instance of the class edit_record; the pattern stored in the grey record has become the pattern of the record Record. Then draws this new grey pattern.

:get_black_record(Instance,~Record)
Asks the record "black_record", instance of the class edit_record, to create a new instance of the class char_record_with_pattern, and to store in it the black pattern and all the information needed by the character mode for creating a new character.

:get_grey_record(Instance,~Record)
Asks the record "grey_record", instance of the class edit_record, to create a new instance of the class char_record_with_pattern, and to store in it the grey pattern and all the information needed by the character mode for creating a new character.

:get_edit_record(Instance,~Record)
Asks the record "black_record", instance of the class edit_record, to create a new instance of the class char_record_with_pattern, and to store in it the black pattern and all the information needed for storing the record in a pattern register and reusing it later safely.

:set_edit_record(Instance,Record)
Record must be an instance of the class char_record_with_pattern, previously saved in a pattern register. Erases on the screen the box and the black pattern, simply by drawing them one more time using an exclusive or operation; saves the record Record into the black record. Draws this new black pattern, and draws the box again.

:set_box_size(Instance,Box_width,Box_height,Box_base)
affects both the black record and the grey record.

:set_box_size(Instance,Box_x,Box_y,Box_width, Box_height,Box_base)
affects both the black record and the grey record.

:get_box_size(Instance,^Box_x,^Box_y,^Box_width,
^Box_height,^Box_base)
takes these values from the black record.

:exchange_black_and_grey(Instance)
draws both records, which has for effect to make them disappear form the screen, gets from the black record the size and the position of the box, puts these parameters in the grey record, exchanges the black and the grey records, and draws both records again.

:add_grey(Instance)
draws the black record for erasing it; asks to the grey record to create an instance of the class char_record_with_pattern and to store its pattern in it; sends this record to the black record, asks it to add the pattern stored in this record to its own pattern, and draws again the modified black pattern.

:clear(Instance,Message)
Message can be 'black', 'grey' or 'all'. Clears the contents of the specified records and clear the specified patterns on the screen. In the case the black pattern is cleared, the box is redrawn in the middle of the screen.

:move(Instance,X0,Y0,X1,Y1,Message)
Message can be 'black', 'grey' or 'all'. Erases the specified patterns on the screen, moves the specified records, and draws again the specified patterns, once moved. The vector of the translation used is simply (X1 - X0,Y1 - Y0).

:move_box(Instance,X,Y)
after a double middle click of the mouse has been detected in this window, this method is called, with the position of the mouse as parameters. Computes from the position of the mouse the translation vector that must be used for moving the mouse; erases the box, sends this translation vector to the black record, in charge of checking the information about the box, and draws the box again, asking its new position to the black record. However, in the case the double middle click was done when the mouse was not just one square outside the box, does nothing.

:reshape_box(Instance,X,Y)
after a single middle click of the mouse has been detected, this method is called, with the position of the mouse as parameters. Computes from the position of the mouse the new size and position of the box, erases the box, and sends this information to the black record for checking. Asks to the record the new size and position of the box, and draws it again. However, in the case the middle click was done when the mouse was not just at one square of distance from the box, does nothing.

:set_brush_color(Instance,Color)

:set_shape_color(Instance,Color)

:next_brush_color(Instance)
has the effect of a circular permutation on the possible values stored in the slot "brush_color".

:next_brush_shape(Instance)
has the effect of a circular permutation on the possible values stored in the slot "brush_shape".

:draw_point(Instance,X,Y)
draws a point, according to the current color and shape of the brush, at the position (X,Y) on the screen, and possibly around.

:draw_curve(Instance)
calls the inherited method :draw_curve/2 with the color 'black'. This method allows the user to draw a pattern by simply moving the mouse on the screen, without having to click on each dot.

:get_cut_record(Instance,X0,Y0,X1,Y1,Record)
asks the black record to save, in a new instance of the class char_record_with_pattern the part of the black pattern which lies inside the rectangle defined by the two opposite points (X0,Y0) and (X1,Y1).

5.5.4 Local Predicate

move(Instance,Vector_x,Vector_y,Message)
(see the instance method :move).

box_move_vector(Instance,X,Y,^Vector_x,^Vector_y,^Status)
computes the position of the column X and of the line Y relatively to the five lines which composes the box. From these relative positions, the position of the dot (X,Y) relatively to the box is determined; if the dot (X,Y) is a square just outside the box, with one segment in common with a line of the box, generates a vector which will move the box of one square, in such a way that the selected square comes inside the box. Otherwise, does nothing.

new_box_size(Instance,X,Y,^Box_x,^Box_y,^Width, ^Height,^Base)
computes the position of the column X and of the line Y relatively to the five lines which composes the box. From these relative positions, the position of the dot (X,Y) relatively to the box is determined; if the dot (X,Y) is a square with one segment in common with a line of the box, generates a new size and position such that the line along the square (X,Y) is moved on the opposite side of the square. Otherwise, does nothing. Conflicts are solved by giving the priority to the horizontal lines, and among the horizontal

lines to the base line.

generate_order(Horz_position,Vert_position,^Position)
Computes the position of a dot relatively to the box from the relative position of the horizontal and vertical lines it belongs to.

new_box_size(Instance,Position,^Box_x,^Box_y,^Width,^Height,^Base)
(see the local predicate new_box_size/9).

get_horizontal_position(Instance,X,^Horz_position)
Computes the position of the vertical line X relatively to the box.

get_vertical_position(Instance,Y,^Vert_position)
Computes the position of the horizontal line Y relatively to the box.

draw_point(Instance,Brush_color,Brush_shape,X,Y)
(see the instance method :draw_point).

CHAPTER 6

TOP LEVEL SYSTEM

6.1 DESIGN CONSTRAINTS

Now, all the tools required for the font editor have been provided. But, for the system to be usable, a control level must be added, in charge of coordinating the existing parts of the system, of interpreting the user inputs and of taking the adequate actions. Are required:

1. an ability to read the input entered by the user, either from the mouse or from the keyboard; to propose temporary menus or windows when necessary; to propose a minimum set of facilities for entering and correcting strings from the keyboard; to be ready to abort an operation at any moment.

2. an ability to dispatch the input to the objects concerned, and to coordinate the communications between these objects.

3. protections against losts of characters or fonts, or mismatch of font names.

6.2 DESIGN COMMITMENTS

Usually, the user will not have to use the keyboard for using the font editor. But he will be asked at times to enter strings, like a font name; under these circumstances, he must use the keyboard. But the windows of the character mode and of the pattern mode are only supposed to receive inputs from the mouse, and there is no reason to modify them. It has been therefore decided to create a new class of windows, specialized in accepting inputs from the keyboard, and in controlling their contents according to the needs of the system. This is the class font_temporary_window. To use a temporary window in such circumstances has several advantages: the user can abort the operation simply by moving the mouse out of the window; the user has a clear impression that the use of the keyboard is exceptional, and that the window has no relation with the rest of the display.

Three classes has been created for dispatching the inputs from the

user and executing the corresponding commands: the classes char_mode, pattern_mode and font_executive. The top level loop (a repeat - fail loop) is an instance method of the class font_executive. It reads an input from the user, and sends it either to an instance of the class char_mode or of the class pattern_mode, according to the current mode, with the notable exceptions of the commands changing the mode, or those used for communicating with outside processes, which are handled directly by the font_executive instance.

For protecting font names from being used for different fonts, another class of objects has been created: the class font_file_manipulator. An object of this class remembers the names of all the fonts already created, by saving them into a file and reading the file when necessary. It can check the new names proposed by the user, and refuses those which already used. This class is inherited by the class font_executive, which is in charge of the interface with the outside world.

The last class defined in the font editor system is the class font_editor. It is used for creating a font editor as an independent process. It inherits the class as_program, and when activated, creates an instance of the class font_executive and calls the top level loop of this instance.

6.3 THE CLASS FONT_TEMPORARY_WINDOW

6.3.1 Description

This class is built from several classes provided by the window system. It inherits the classes sash; as_input; as_mouse_input; as_output; the class as_temporary_window, which provides to its instances the ability of disappearing as soon as the mouse is no longer inside them; and the basic window class user_window.

An instance of this class has several methods which display a message and filters the user inputs into the expected answer format. Each of these methods provides several ways for aborting the execution of the current command to the user. Of course, putting the mouse out of the window will always have this effect.

6.3.2 Instance Methods

:ask_for(Instance,Message,~String)

writes on the window screen the string Message, and waits for the user to enter a string from the keyboard. If the user asks for aborting, the variable String is instantiated to the atom 'abort'. Otherwise, it is instantiated to a string of characters, which are currently limited to the letters from "a" to "z" and from "A" to "Z", to the figures, and to the character "_". The size of the string is limited to 256 double bytes.

:ask_for_list(Instance,Message,^List)

writes on the window screen the string Message, and waits for the user to enter a list of integers from the keyboard. If the user asks for aborting, the variable List is instantiated to the atom 'abort'. Otherwise, it is instantiated to a stack list of integers.

:ask_confirmation(Instance,Message,^Answer)
writes on the window screen the string Message, adds the string " (Y/N) " after it, and waits for the user to answer by yes or by no. If the user enters either "y", "Y" or a carriage return, Answer is instantiated to the atom 'yes'; otherwise to the atom 'no'.

6.3.3 Local Predicates

ask_for(Instance,Message,^String)
creates a buffer, as a new double byte string of length 256; shows the window, writes the message Message in it, filters the input; if the input is an order to abort, instantiates String to the atom 'abort'; otherwise instantiates String to the contents of the buffer.

ask_for(Instance,Message,Buffer,^Length,Pointer)
reads the input from the user. The :read method fails only when the user puts the mouse outside the window; in that case, instantiates the variable Length to the atom 'abort' and stops. Otherwise, if the input from the user is a carriage return code, instantiates Length to the value of Pointer, and stops. If the input is a Control-D code, clears the display of the window, writes the message Message again, resets the pointer to 0, and calls itself recursively. If the input is a delete code, erases the last character if there is any, decreases the pointer by 1 if it has a positive value, and calls itself recursively. If the input is a letter, a figure, or the underscore character code, writes the corresponding character on the screen, adds the code in the string Buffer at the position Pointer, increases the pointer by 1 and calls itself recursively; otherwise, simply calls itself recursively, ignoring the input. Stops when the pointer is equal to 256.

ask_confirmation(Instance,Message,^Answer)
(see the instance method :ask_confirmation).

ask_for_list(Instance,Message,^List)
shows the window, writes the message Message, creates a buffer, as an instance of the class list; filters the inputs; if the buffer is returned empty, instantiates List to the atom 'abort'; otherwise instantiates List to the contents of the buffer.

fill_buffer(Instance,Buffer)
Buffer must be an instance of the class list. Reads the next number from the user input; if it receives an order to abort,

empties the buffer Buffer and stops; if it receives an order to stop, adds the last number read from the input in the buffer and stops; otherwise, adds the last number read from the input into the buffer, and calls itself recursively. Stops when the buffer contains ten integers.

read_number(Instance,^Number,^Message)
creates a new buffer, as a double byte string of length 5; stores the next integer from the user inputs into the new buffer; if an order to abort is transmitted, stops; otherwise converts the contents of the buffer into an integer, instantiates the variable Number to this integer and stops.

read_number(Instance,Buffer,^Message,^Length,Pointer)
reads the input from the user. The :read method fails only when the user puts the mouse outside the window; in that case, instantiates the variable Length to the atom 'abort' and stops. Otherwise, if the input from the user is a carriage return code, instantiates the variable Message to the atom 'abort' if the pointer is equal to 0, or to the atom 'last' otherwise, and stops. It that last case, instantiates the variable Length to the value of the pointer. If the input is a delete code, erases the last character if there is any, decreases the pointer by 1 if it has a positive value, and calls itself recursively. If the input is a figure, writes the corresponding character on the screen, adds the code of the figure in the string Buffer at the position Pointer, increases the pointer by 1 and calls itself recursively. If the input is the code of one of the allowed separators: ",", ";", ".", "/", "\", "¦", "$", "@" or "#", writes the character on the screen, instantiates the variable Message to the atom 'not_last', the variable Length to the value of the pointer and stops, unless the value of the pointer is 0, where in that case it instantiates the variable Message to the atom 'abort' and stops. Otherwise, ignores the input,and simply calls itself recursively.

convert_to_number(String,^Result,Integer,Length,Pointer)
reads the element of position Pointer in the string String; extracts the digit of code this element, adds it to ten times the value of Integer; puts the result in Integer, increases Pointer by 1, and calls itself recursively. When Pointer is equal to Length, instantiates Result to the value of Integer, and stops.

6.4 THE CLASS FONT_FILE_MANIPULATOR

6.4.1 Description

This class provides to the class font_executive, which inherits it, the interface methods with font files. Since the class as_font_file already provides the basic methods for saving a font into a file or for loading a font from a file, the main mission of an instance of this class is to know whether a given string is already used as a font name or not. It keeps into a file the list of the names of the font which have been created in the current machine. When created, an instance of this class loads these names from a file into a list, kept as an attribute. Each time a new font is created, it checks if the name proposed by the user is acceptable, and, in that case, accepts the new name; it stores it both in the list and in the file, when the new font is itself saved into a file. It uses the same fixed directory as a instance of the class as_font_file.

6.4.2 Instance Attributes

1. font list: contains an instance of the class list, in which the list of the font names is stored. A font name is a string of characters, not a atom.

2. directory path: fixed attribute, initialized to the path of the directory currently used for fonts: ">sys>user>font".

3. font list file path: fixed attribute, initialized to the name of the file used for storing the names of the fonts: "fonts".

4. number of fonts: number of font names recorded in the file containing the font names. Computing the value of this attribute corresponds to unnecessary lines of code, since the class list provides the instance method :count($list,^N), which computes the desired parameter, when used with the list "font_list".

5. file length: length, in number of double bytes, of the file used for storing the names of the fonts. It is stored in the file itself. Since one double byte is used as a separator after each font name in the file, the value of this attribute is equal to the sum of the number of characters which composed each font name, augmented by the number of font names recorded, augmented by 1. The "1" corresponds to the double byte in which this parameter is stored.

6.4.3 Instance Methods

:load_font_list(Instance)
computes the path name of the file in which the font names are saved, from the values stored in the slots "directory_path" and "font_list_file_path"; reads the file, puts its contents

in the slot "file_length" and in a stack list, computes the value of the slot "number_of_fonts", and copies the contents of the stack list into the list "font_list".

:save_font_list(Instance)
computes the path name of the file in which the font names are to be saved, and, if the list "font_list" is not empty, saves its contents into the file, as well as the contents of the slot "file_length".

:new_font(Instance,Font_name,~Font)
if the name Font_name is found in the list "font_list", fails; otherwise, creates a new instance of the class edited_font, of name Font_name. Does not remember the name of this font before the font is saved in a file.

:save_font(Instance,Font)
saves the font Font in file, and asks for its name; if this name is already recorded in the list "font_list", does nothing; it corresponds to the case of a font that has been modified but not created. Otherwise, adds the name to the list "font_list", modifies the values of the slots "number_of_fonts" and "file_length" consistently, and saves into a file the names and the file length.

:load_font(Instance,Font_name,^Font)
if the name Font_name is not found in the list "font_list", fails; otherwise, creates a new instance of the class edited_font, of name Font_name, and loads the contents of this font from file.

6.4.4 Class Method

:create(Class,Instance)
creates a new instance of this class and loads in its slots the contents of the file containing the list of the names of the fonts.

6.4.5 Local Predicates

registrate_name(Instance,Name)
adds the name Name to the list "font_list", increases the value of the slot "number_of_fonts" by 1, and increases the value of the slot "file_length" by 1 plus the number of characters of the name Name, except when the name Name is the first to be saved in the list. In that case, sets the value of the slot "file_length" to be equal to 2 plus the number of characters of the name Name. The difference of 1 comes from the fact that the first double byte recorded in the file is reserved for recording the length of the file.

load_font_list(Instance,^List,Path_name)

opens the file of path Path_name; reads the first double byte recorded in the file and saves it into the slot "file_length"; copies the rest of the file in a string; then puts the font names contained in this string in a list, counts their number, and saves this number into the slot "number_of_fonts". If the file cannot be opened, simply instantiates List to [] and stops.

put_into_list(^Font_names_list,String,Length,^Nb_of_fonts,
Counter,Pointer)
Computes the position of the first character of the next font name stored in the string String; puts in the list Font_name_list the substring of the string String beginning at the position Pointer and ending at two double bytes before the first character of the next name; increases Counter by 1, puts Pointer at the position of the beginning of the next name, and calls itself recursively. When Pointer is equal to Length, the length of the string String, instantiates Nb_of_fonts to the value of Counter, and stops.

end_of_record(String,Length,Pointer,^Next_name_position)
if the double byte stored in the string String at the position Pointer is the code of the character delete, which is used as a separator between font names in the file, instantiates Next_name_position to Pointer + 1; otherwise increases Pointer by 1 and calls itself recursively. When Pointer is equal to Length, the length of the string String, instantiates Next_name_position to Length and stops.

save_font_list(File_length,Font_names_list,Path_name)
makes a new binary file of path name Path_name, creates a string of length File_length, puts at the first position in this string the number File_length, and copies after it the strings contained in the list Font_names_list, one after the other, using the code of the character delete as a separator at the end of each name; then copies the contents of the string in the file.

put_into_string(Font_names_list,String,Pointer)
removes the next string from the list Font_names_list, copies it in the string String at the position Pointer, and adds a the code of the character delete after it; increases the pointer Pointer of 1 plus the length of the next string, and calls itself recursively. Stops when the list Font_names_list is empty.

registrated_name(Instance,String)
loads from file the names of the fonts into the slot "font_list", and checks if the string String belongs to the list of the font names. If yes, succeeds; if not, fails. Since it is not possible to unify two strings which are equal but not saved in the same memory location, the strings must before be converted into stack lists, in which the codes of their characters are stored. Stack lists can be unified if their contents are identical, even if they do not share the same memory space.

font_member(List_of_codes,List_of_strings)
succeeds if there is one string in the list List_of_string such that the list of the codes of its characters is equal to List_of_codes; fails otherwise.
This predicate contains a logical bug since the method string_to_list of the class symbolizer cannot be used with the result already instantiated. The line of code:
:string_to_list(#symbolizer,List,String), !;
must be replaced by the line:
:string_to_list(#symbolizer,L,String), !, L = List;

path_name(Directory,File_name,~Path_name)
(see the local predicate path of the class as_font_file).

6.5 THE CLASS CHAR_MODE

6.5.1 Description

This class inherits the class char_mode_display. Its only job is to provide methods required by the top level for executing the commands entered by the user. Since the different menus and windows that composed the character mode display already dispose of basic methods for executing the commands entered by the user, the main responsibility of an instance of this class is to coordinate the actions of the various objects which compose the character mode display.

It coordinates the scrolls of the window displaying the character patterns, and of the windows displaying the 8 lower bit part and the 8 higher bit part of the character codes respectively.

It also coordinates the transfer of patterns, from the windows displaying the black and the grey patterns of the pattern mode, to the window displaying the character patterns of the current font, as well as the transfer effected in the opposite direction.

And it assures the communication of data between the top level and the windows and menus of character mode.

6.5.2 Instance Methods

:horz_scroll(Instance,Message)
Message can be either 'left' or 'right'. Sends the message Message to both the horizontal window displaying the lower 8 bit part of the code and to the window displaying the character patterns.

:vert_scroll(Instance,Message)
Message can be either 'up' or 'down'. Sends the message Message to both the vertical window displaying the higher 8 bit part of the code and to the window displaying the character patterns.

:code_selected(Instance,Input,^Answer)
If the input Input is a left click of the mouse, computes the position of the mouse, selects the code Code corresponding to this position in the window displaying the character patterns, and, if either another code, Old_code, or one of the two windows displaying the grey and the black patterns of the pattern mode has been selected before, instantiates the variable Answer respectively to the terms load(Old_code,Code), load_black(Code) or load_grey(Code). Otherwise, instantiates Answer to the atom 'select' if no code and no window was selected before, and to the atom 'abort' otherwise, which corresponds to the case where Code and Old_code are equal.

It is really pleasant to use the facilities provided by a Prolog like language for building terms and using them as messages. But in that case, the message is sent to the top level, which has to understand it, and then to send back orders an instance of this class for executing the order. This can seem an unjustified overhead. The reason why the orders cannot be fully executed inside the character mode is that the user may want to assign a pattern to a code without having created a font before. In that case, the font editor has to propose to the user to create a new font, and to check if the name proposed by the user for the new font is not already used for another font. This means an interaction with objects outside the character mode, which therefore must be handled by the top level.

:black_selected(Instance,Input,^Answer)
If the input Input is a left click of the mouse, selects the contents of the window displaying the black pattern of the pattern mode; if this pattern was not already selected, and if a code Code was selected in the window displaying the character patterns of a font, instantiates the variable Answer to the term load_char(Code). If no code is currently selected, instantiates Answer to 'select'; otherwise instantiates Answer to the atom 'abort'.

It would be far better that this method does not send messages to the top level, but execute itself the corresponding order, since, in that case, there is no reason for the top level to do it.

:get_record(Instance,Color,^Record)
Color can be either 'black' or 'grey'. Gets the contents of the window displaying either the black or the grey pattern of the pattern mode.

:set_record(Instance,Color,Record)

:grey_selected(Instance,Input,^Answer)
(Similar to the method :black_selected).

:set_font(Instance,Edited_font)
sends the font Edited_font to the window in charge of displaying the character patterns of a font.

:get_font(Instance,^Edited_font)

:draw_char(Instance,Code)
draws, in the window in charge of displaying the character of a font, the pattern of the character of code Code, in the currently displayed font.
This method appears twice in the source program; once is probably enough.

:refresh(Instance,Color)
Color can be either 'black' or 'grey'. Clears the display of the window displaying the black or the grey pattern of the pattern mode, and draws its display again.

6.5.3 Local Predicates

code_selected(Instance,Input,^Message)
(see the instance method :code_selected).

code_selected(Instance,Old_code,X,Y,^Answer)
Old_code must be instantiated to the previous code selected in the window displaying a font, or, if there is none, to the atom 'nil'. Selects the code pointed by the mouse in (X,Y) in the window displaying a font.
In the case Old_code is equal to nil, asks to this window which code has been selected, and asks to the black and the grey window if their contents have been selected or not. If the contents of the black window have been selected, instantiates Answer to the term load_black(Code); otherwise if the contents of the grey window have been selected, instantiates Answer to the term load_grey(Code). Otherwise, instantiates Answer to the atom 'select'.
In the case Old_code is a code previously selected, asks which code has been selected. If the answer is 'nil', it means that Old_code has been selected twice. In that case, instantiates Answer to the atom 'abort'. Otherwise, instantiates Answer to the term load(Old_code,Code).

black_selected(Instance,Input,^Answer)
(See the instance method :black_selected).

grey_selected(Instance,Input,^Answer)
(See the instance method :grey_selected).

6.6 THE CLASS PATTERN_MODE

6.6.1 Description

This class inherits the class pattern_mode_display. Its only job is to provide methods required by the top level for executing the commands entered by the user. It plays, for the pattern mode, the role that the class char_mode plays for the character mode.

It gives to the top level the access to the black and the grey patterns displayed in the window used for editing patterns; this is the only information that is transmitted to the character mode when the mode is changed. This information must be communicated to the top level, which is in charge of coordinating the changes of mode.

And it executes itself all the orders the user enters in the pattern mode, except those orders which need to communicate with the character mode, which are executed in the top level, or with the help of the top level.

6.6.2 Instance Methods

:get_record(Instance,Color,~Record)
Color can be either 'black' or 'grey'. Asks to the editing window the pattern displayed either in black or in grey, in a format adapted to the character mode.

:set_record(Instance,Color,Record)
Color can be either 'black' or 'grey'. Sends to the editing window a pattern it must display in the color Color. The record Record is in a format adapted to the character mode.

:brush_menu_order(Instance,Input)
The corresponding menu proposes to the user to set the color or the size of the brush, which is nothing else than the mouse that the user uses like a brush for editing character patterns. See the class pattern_mode_display and edit_pattern_window for details. This method simply sends the message Input to an instance of the class edit_pattern_window.

:global_menu_order(Instance,Input)
the corresponding menu proposes various functions as to display the black pattern in real size, to change the grid size, or to clear both the black and the grey records. See the class pattern_mode_display for details about these commands.

:black_menu_order(Instance,Input)
the corresponding menu proposes various functions for helping the editing of the black pattern. The execution of these functions consist in waiting for supplementary input from the user, if required, and calling the suitable method of an instance of the editing window.

:grey_menu_order(Instance,Input)
Similar to the method :black_menu_order.

:parameters_menu_order(Instance,Input)
for any input, makes the menu displaying the size of the box display the current size of the box.
When the shape of the box is modified, the new parameters are not automatically sent to this menu. It would be much better to have the size of the box automatically sent to this menu each it is modified. It is much clearer for the user, and does not cost anything to the programmer, since all the necessary methods exist already.

:pattern_window_order(Instance,Input)
interprets the mouse clicks from by the user in the window used for editing patterns, and executes the corresponding commands: either writing dots, moving the box, modifying the shape of the box, modifying the color or the shape of the brush.

:registers_window_order(Instance,Input)
interprets the mouse clicks from by the user in the window containing registers, and executes the corresponding commands: loading the black pattern into the register pointed by the mouse, or loading the contents of the register pointed by the mouse into the editing window as the black pattern, saving the previous black pattern in a special register; or clearing all the registers.

6.6.3 Local Predicates

brush_menu_order(Instance,Input)

global_menu_order(Instance,Input)
uses the local predicates wait_for_vector, ask_for and ask_for_list for asking more information to the user when it is required. See these local predicates for details.
Otherwise, only the case where Input has been instantiated to a term of the form: set_sample(Edited_font) has some interest. Since an instance of this class has no connection with the character mode, and thus with the font currently being edited, which is kept inside the character mode, it cannot not have executed the order "SET SAMPLE" by itself. To help it, the top level filters the inputs, and when the message "SET SAMPLE" is encountered, it asks the character mode to send to it the font currently edited, and sends this font to the pattern mode inside the message set_sample. The possibility of using terms in ESP is really convenient. It allows the programmer to add a parameter only used in some cases, by modifying only the parameter itself in these cases, not the predicate.

black_menu_order(Instance,Input)
uses the local predicate wait_for_vector, for asking more information to the user when it is required. See this local predicate for details. Note that the order "DRAW" corresponds to the instance method :draw_curve of the class

edit_pattern_window.

grey_menu_order(Instance,Input)
uses the local predicate wait_for_vector, for asking more information to the user when it is required. See this local predicate for details.

parameters_menu_order(Instance,Input)
(See the instance method :parameters_menu_order).

pattern_window_order(Instance,Input)
See the instance method :pattern_window_order. In the present version of the font editor, the right click of the mouse is reserved as an interruption key. It is therefore not currently possible to change the size of the brush by a right click in the editing window. This must be fixed, either by authorizing an interruption only when the right click is done in another window that the editing window, or by suppressing this interrupting facility. The second solution is probably better. The interrupting facility was only used for helping the debugging of the program.

registers_window_order(Instance,Input)
if the input is a left click, checks if the register pointed by the mouse is the first register on the left, or, if it is not, if it is empty or full; asks to the editing window a copy of the black pattern, and calls with all this information the local predicate automatic_registers. If the input is a middle click, loads a copy of the black pattern into the register pointed by the mouse, erasing the contents of this register, except when there is no black pattern or when the register selected is the one on the left. If the input is a double middle click, erases the contents of all the registers. Otherwise, does nothing.

automatic_registers(Instance,Status,Record,X,Y)
Status can take three different values: 'protected', if the register pointed by the mouse at the position (X,Y) is the first on the left; 'full' if the register pointed by the mouse is not the first on the left and contains a pattern; 'empty' otherwise. Record is a copy of the black record displayed in the editing window, or, if there is none, the atom 'nil'.

if the status is 'protected', and if Record is equal to 'nil', sends the pattern displayed in the first register to the editing window as its new black pattern. If Record is not equal to 'nil', exchanges the black record and the pattern displayed in the first register.

If the status is 'empty', and if Record is not equal to 'nil', loads the record into the selected register. If the status is 'full', and if Record is equal to 'nil', copies the pattern displayed in the register into the editing window, as its new black record. If Record is not equal to 'nil', does the same first, and then saves Record into the first register. Otherwise, does nothing. For more details, see the class pattern_registers_window.

check_grid_size(String,^Grid_size)
this predicate is used by the local predicate global_menu_order when the input is equal to 'scale'. The user is asked to enter a new grid size. The result of this input is stored in the string String, and has to be interpreted by this predicate. The variable String can be instantiated either to a double byte string or the atom 'abort'. If String is a string of length 1 or 2, containing only figures as characters, instantiates Grid_size to the corresponding integer; otherwise instantiates Grid_size to the atom 'abort'.

check_grid_size(String,String_length,^Grid_size)
(see the local predicate check_grid_size/2).

load_record_list(Record_list,List_of_codes,Edited_font)
This local predicate is used by the local predicate global_menu_order when the input is 'set_sample'. Record_list must be an instance of the class list, List_of_codes a stack list of double bytes integers and Edited_font an instance of the class edited_font. For each code found in the list List_of_codes, tries to get the corresponding character record in the font Edited_font. If it finds one, adds it to the list List_of_records, otherwise goes to the next code. Stops when the list List_of_codes is empty.

ask_for(Message,~String)
creates an instance of the class font_temporary_window, calls the instance method :ask_for of this class, and kills the window.

ask_for_list(Message,~String)
creates an instance of the class font_temporary_window, calls the instance method :ask_for_list of this class, and kills the window.

wait_for_vector(Instance,^X0,^Y0,^X1,^Y1,^Message)
this local predicate is called by several other ones of this class. It waits until the user clicks the mouse in the editing window twice, for entering the coordinates of two points: (X0,Y0) and (X1,Y1). Only the left click is accepted. If the user does what is expected, Message is instantiated to 'normal'; otherwise Message is instantiated to 'abort', and some of the other variables may not be instantiated.

These two points can be the source and the destination points used for translating patterns , or the two opposite corners of a rectangle used for cutting patterns in the editing window.

As it is, this predicate is only half usable, because it does not show that it reacts to the user inputs. It must be modified in such a way that the user can always see on the screen what is going on. The user has a very limited short time memory, which should not be encumbered by such irrelevant things as whether or not he has already click the mouse twice.

6.7 THE CLASS FONT_EXECUTIVE

6.7.1 Description

This class is the top level class. It only inherits the class font_file_manipulator, which provides methods for communicating with the file system. When a font editor is created, an instance of this class is created, and the top level loop is called. When created, an instance of this class creates three objects: a superior window, an instance of the class char_mode and an instance of the class pattern_mode. The superior window is declared superior to all the windows and menus of the character mode and of the pattern mode (see the class as_font_display). It keeps as attributes these three objects.

The role of the superior window is to allow an instance of this class to read through it, in a single instruction, each input from the user, with the window or the menu through which the input was done. Knowing from where the inputs came allows to dispatch them easily to the objects concerned.

Most of the orders are simply dispatched to the instances of the class char_mode and pattern_mode; those which involve both the pattern mode and the character are executed directly by a instance of this class. There are however a few exceptions (see below and see the classes char_mode and pattern_mode).

6.7.2 Instance Attributes

1. superior: contains the superior window through which the inputs are read by the top level loop.
2. char_mode: contains an instance of the class char_mode, created just after the creation of this instance.
3. pattern_mode: contains an instance of the class pattern_mode, created just after the creation of this instance.
4. display_mode: this slot can contain either the atom 'char_mode' or the atom 'pattern_mode'. It is used for remembering the current mode.

6.7.3 Instance Methods

:show(Instance)
displays the current mode.

:execute_orders(Instance)
is the top level loop. It simply repeats the execution of the method :execute_order (without "s") until it succeeds.

:execute_order(Instance)
is the top level step. It reads the next input and the window or menu from which it came through the superior window "superior". If the input is a right mouse click, succeeds; otherwise tries to execute the order corresponding to the input: if it succeeds to do so, then fails; otherwise succeeds.
The interrupt facility offered by the success of this method when a right mouse click is entered, was designed for helping the debugging of the program. Since it is far from being convenient for the user, who can use it accidentally, it may be better to suppress it in a more user-friendly version.

:kill(Instance)
saves the currently edited font, if any, in files; saves the list of the names of the existing fonts in file; and kills all the windows and menus that were created at the creation of the instance.

6.7.4 Class Method

:create(Class,~Instance)
creates a new instance of this class; creates a superior window, an instance of the class char_mode, and an instance of the class pattern_mode, with superior window this new superior window. Then saves these three objects into the corresponding attributes, shows the superior window, draws both the character mode and the pattern mode displays, and shows the current mode, which is the character mode.

6.7.5 Local Predicates

execute-order(Instance,Input_channel,Input,Mode)
Input_channel must be instantiated to the window or the menu through which the user entered the input Input. Mode must be instantiated to the current mode. This local predicate asks to the input channel its name, recorded in an attribute when it was created (see the class as_font_display), and dispatches the input according to the mode.

execute_char_order(Instance,Input_channel_name,Input)
If the input channel is the menu "command_menu" of the character mode, calls the predicate command_menu_order. In that case, the corresponding input is an order concerning the interaction of the character mode with another object: the pattern mode, a file, a new font. It is therefore to an instance of this class to execute the order. It is what the local predicate command_menu_order is doing.
In the case the input channel is either the window displaying the character patterns of a font, or the black or the grey pattern of the pattern mode, the input is sent to the character mode for a preprocessing, which determines the

nature of the operations to be executed. These operations are then executed from an instance of this class by eventually calling character mode methods. They could be executed inside the character mode, except for the local predicate loading_in_font, which calls the local predicate create_font, when the user tries to assign a pattern to a code, before a font has been loaded or created. The local predicate create_font must be executed in an instance of this class, since it has to know how to prevent the occurrence of name conflicts between fonts. See the class char_mode.

The other inputs, which concern only the character mode, are simply sent to the character mode "char_mode", and executed by it.

execute_pattern_order(Instance,Input_channel_name,Input)

If the input channel is the menu "ouside_menu" of the pattern mode, calls the predicate outside_menu_order. In that case, the corresponding input is an order concerning the interaction of the pattern mode with another object: the character mode, or a file providing information to the user on how to use the system (this is not implemented yet). It is therefore to an instance of this class to execute the order.

In the case the input channel is the menu "global_menu" of the pattern mode, and the input is the order 'set_sample', asks to the character mode the font which is currently edited, and sends the message: set_sample(Edited_font), to the pattern mode, which can then execute the order.

The other inputs, which concern only the pattern mode, are simply sent to it and executed by it.

command_menu_order(Instance,Input)

If the input is 'exit', calls the method :kill and fails, to stop the top level loop.

If the input is 'disc', creates and shows a temporary menu which proposes to the user either to save the current font or to load another font. If the user chooses to save the current font, shows a temporary window displaying the name of this font, and asks for confirmation. If the user chooses to load a font, asks to him, through a temporary window, the name of the font to be loaded. Asks a new name until the name entered corresponds to an existing font, or until the user aborts by moving the mouse outside the temporary window.

If the input is 'edit', changes the mode to be the pattern mode, shows the display of the new mode, asks to the black and the grey windows of the character mode the patterns they were displaying, and sets these patterns as the black and the grey patterns of the editing window in the pattern mode.

create_font(Instance,~Font)

asks to the user, through a temporary window, a new name for a font; asks a new name until the user enters a name which is not already used for a font, or until he aborts the operation by moving the mouse out of the temporary window. When an acceptable name is received, creates an new instance of the class edited_font, of name that name.

outside_menu_order(Instance,Input)
If the input is 'char_mode', changes the mode to be the character mode, shows the display of the new mode, asks to the editing window its black and grey patterns, and sets these patterns as the patterns displayed by the black and the grey windows of the character mode.
If the input is 'help', displays a message of encouragement to the user.

ask_for_font_name(~String)
creates an instance of the class font_temporary_window, displays a message asking for a font name and waits for an answer. Calls the instance method :ask_for of the class font_temporary_window. Similar to the local predicate ask_for of the class pattern_mode.

ask_for_another_name(~String)
The only difference with the predicate ask_for_font_name is the message displayed in the temporary window.

loading_in_font(Instance,Record,Code)
asks to the character mode the currently edited font; if there is none, asks the user to create a new one; if he does not want, stops; otherwise, stores in the font the record Record, at the code Code, clears and draws again the display of the character mode, incorporating the new pattern.
It is required to redraw the character mode display each time a new pattern bigger than the ones which are already recorded is saved into the font (see the classes char_mode_display and font_display_window). But it is not required when the new pattern is not bigger. The present implementation does not optimize, and redraws the character mode each time a character is created with a new pattern, regardless to its size.

copy_in_font(Instance,Record,Code)
is used for assigning to a code a pattern already assigned to another code in the current font. Asks to the character mode the currently edited font; if there is none, is supposed to stop: but there is a bug left in the program, a comma appearing instead of a semi-column. This must be fixed.
Once the bug is fixed, this local predicate will behave as described in the following lines. If there is no font currently being edited, stops; if Record is equal to 'nil', deletes the previous pattern assigned to the code Code; otherwise, if the code Code is equal to 0, assigns the record Record as the new pattern of this code, and redraws all the display of the character mode, since the pattern of the code 0 is used as a default pattern. Otherwise, assigns the record Record to the code Code, and simply draws the pattern contained in this record. There is no need to redraw entirely the display in this case, since the pattern is not entering the font.

6.8 THE CLASS FONT_EDITOR

6.8.1 Description

This class inherits the class as_program, and is used when the user wants to run the font editor as an independent process. When an instance of this class is activated, it creates an instance of the class font_executive, and calls the top level loop of this instance.

For a more user-friendly interface, it would be better to inherit the class as_e_program and to add an entry "font_editor" in the system menu.

6.8.2 Instance Method

:goal(Instance)
creates an instance of the class font_executive, and calls the method :execute_orders of this instance.

CHAPTER 7

PROPOSED IMPROVEMENTS

7.1 INTRODUCTION

In this last part a set of improvements of the current version of the font editor are described. Most of them are easy to implement by patching the existing program. These improvements have been suggested after several days of practice with the font editor and their main concern is to improve the user interface of the system. The possibility of implementing metafont-like facilities is also briefly suggested.

7.2 ALGORITHM OPTIMIZATION

7.2.1 Pattern Mode Display

In the present state, each time a new black pattern is loaded into the pattern editing window, the old one is erased entirely and the new one is drawn afterwards. It will be much better for the user's eye, and slightly quicker if these two operations are done at the same time, modifying only the squares that must be modified. This can be done without losing too much of the modularity of the program, by giving the hash table in which the old pattern has been stored as a parameter of the method in charge of drawing the new pattern.

Similarly, when the black and the grey patterns are exchanged, both patterns are erased, exchanged and then redrawn in the present implementation. It would be twice as quick to use an exclusive or operation for adding a dark grey square on all the squares of the black pattern and on all the squares of the grey patterns, except the common ones. The pattern corresponding to the dark grey square can be stored as an attribute of the window, the just as the black and the light grey square patterns are stored now. This trick could be also used when the grey pattern is added to the black one.

The algorithm used for writing the grid should also be optimized. It is slow, it does not take advantage of the regularity of the grid, and it is linear in its number of lines. Since there are usually more than 600 lines to write, it could be advantageous to write a logarithmic algorithm instead.

7.2.2 Character Mode Display

When a new pattern is assigned to a code, the window display with the character patterns is entirely redrawn. But this is only necessary when the new pattern is bigger than any of the patterns already drawn. On the other hand, when the widest or the highest pattern is deleted, the display is not redrawn. Both these features are somewhat inconvenient.

When a default pattern is assigned to the code 0, this default pattern is drawn for all the codes which have not be assigned a pattern. The algorithm used for doing this is linear in the number of codes displayed and far too slow. It may be better to fill the window screen quickly, with a logarithmic algorithm, and then add the other patterns. The speed of execution can also be improved by drawing the image of the window screen in a bitmap area first, and then transferring the result to the screen instead doing this operation for each pattern separately.

7.3 FUNCTIONAL IMPROVEMENTS

7.3.1 General Improvements

The user interface could be significantly improved if the limitations of the human short time memory is taken into account. Presently, it is not. The user has often to keep in mind the inputs he entered and to figure out by himself what the program is doing. It is not at all what he wants to do. He wants to concentrate on the font he is editing and wants the editor to help him, not to disturb him.

First of all, in the present state, it is not possible for a program to read an input from a window which is not selected. Therefore, when the user wants to communicate with the window, he has to check if this window is selected or not, and if it is not, he must select it. This is rather confusing, since the same input in the same window can either select the window or send a message to the font editor. The next release of the window system will allow the user to read from a window which is not selected, and this problem will then be easily fixed. But since now it is possible to improve the interface somewhat, by choosing names that are easy to understand for the windows and menus used by the font editor. These names will be displayed at the bottom of the screen. The presently used names (the names of the class of objects to which the menus or the windows belong) are not easy to understand.

When the user enters commands which are not immediately executed, either because they are slow or because they require supplementary information from the user, it is often not possible to know from the display which command has been entered. If the user is interrupted in his work for a while, he is likely to forget this command and be embarrassed when coming back to the font editor. Two features can be easily added for handling this problem. The first one is to blacken the currently executed command on the display, until the command is executed. This facility is provided by the window system. The second concerns only the pattern mode display.

7.3.2 Pattern Mode Display

When using the pattern mode, the user often has to specify two squares on the screen of the window editing patterns by clicking the mouse. The user may forget whether or not he has already selected a square. So, the first square selected should be shown by drawing a smaller temporary square inside it, or by any other means.

The color and the shape of the brush currently in use must also be displayed on the screen. The simplest way is to blacken the corresponding command in the brush menu.

7.3.3 Character Mode Display

What the user usually wants to do in this mode is to assign a pattern to a code. Often, he knows precisely to which code he wants to assign a pattern and finds it really surprising to be obliged to compute the lower 8 bits part and the higher 8 bits part of the code, to convert them to hexadecimal notation, and to scroll several pages of display before being able to do it. Moreover, the two menus proposing the scrolling commands are in the two opposite upper corners of the screen, obliging the user to move the mouse from one side of the screen to the other too often.

It could be much better to put all the scrolling commands at the top left or top right corner of the screen and to use the other corner for another window. In this new window, two codes should be displayed: the smallest code currently displayed and the selected code, if any. The user should be allowed to choose the smallest code to be displayed by entering it directly from the keyboard in the window.

When the user selects a code in the window displaying a font, a rectangle appears around the corresponding position in the screen. But when he selects the contents of any of the two windows at the bottom right corner of the screen, nothing shows it. It should be fixed, by drawing a rectangle inside the window or by any other means.

The user should not have to remember which of the two windows at the bottom left corner of the screen corresponds to the black pattern or to the grey pattern of the pattern mode. The window corresponding to the grey pattern could display the pattern not in black on white, but in black on light grey.

The user should not have to remember the exact names of the font already created. When he wants to load a font to modify it, he should be given a menu displaying the names of the different fonts currently available. This can probably be done by patching the file manipulator source code.

7.4 NEW FEATURES

7.4.1 Pattern Mode

Some commands proposed by the menus have not been implemented yet. They can be implemented on the existing system by simply adding one method for doing the work, and another one for coordination. They are: the "ROTATE" command, which rotates the black pattern at an angle of 90, 180, or 270 degrees, around a square or an intersection of squares chosen by the user; the "REFLECT" method, which reflects the black pattern relative to a line inclined at 0, 45, 90, or 135 degrees, specified by the user.

Another function: the "STRETCH" function could be added in the same way. This function stretches or reduces the black pattern vertically or horizontally. The user specifies two squares on the same line or the same column. From the direction (left to right or right to left; up down or down up) the command determines if it has to stretch or to reduce the pattern. The lines or columns are added or suppressed at uniform intervals. A line suppressed is added to the previous line. A line added is copied from the previous one.

Two other functions may be also useful to implement. One, the "INVERSE" function, would exchange all the black squares in white and all the white squares in black inside a rectangle specified by the user. The other, the "OR" function, would add the grey pattern to the black pattern by an exclusive or operation.

There are only three different shapes of brush proposed currently, and the names of the commands do not describe them fully. It would be much better to display the shapes themselves rather than words describing them. It takes far less space, and it is straightforward to understand. More shapes can be proposed, and the present version of the font editor can be used to edit the corresponding character patterns to display in the brush menu. And it could be possible to let the user define his own brush shapes and to customize his font editor.

7.4.2 Character Mode

The most important functions that should be added in this mode are facilities to load the fonts from or to save them in floppies the fonts. This can be done easily by patching the existing source code handling the interface with disc files.

7.5 ADDING METAFONT FACILITIES

7.5.1 Motivations

Metafont (1) has several very interesting features which could be implemented in a font editor, even if it is more oriented towards typesetting than towards bitmap displays, the latter having a much bigger dot size.

Metafont allows one to define character fonts by specifying the key dots of a pattern, and the value of several parameters such as the scale, or the thickness of the lines. It computes cubic curves itself to interpolate between the specified dots. The programmer can specify the slopes of these curves at the specified dots if he wants to.

By changing the values of the parameters, the programmer can create various kinds of fonts from the same skeletons. Moreover, the rounding of the cubic curves to points of integer coordinates is optimized, which gives a nice appearance to the patterns even with a large dot size.

Metafont provides a really natural way to create character patterns. To implement it, even partially, on the present font editor would require a lot more work. But let us dream for a while, and imagine how easy it would be for the user to edit patterns if such facilities were provided.

The user would just have to think of the proportions of the pattern he wants to create, and to enter a few essential dots of this pattern with the mouse. Then entering the metafont mode, he will be asked to specify the order the cubic interpolation has to follow, as well as the slopes of the cubic curves at each of the principal dots. Default values will be provided and the slopes can be entered in a visual way, by moving a line with the mouse. The program would perform the interpolation. If the result is not satisfactory, the user could change some parameters, come back to the skeleton to modify some of its dots, and try again. This would allow the user to concentrate on the pattern itself, and not, as is the case with traditional font editors, on the regular rounding or thickness of the pattern, which is very hard to get perfect.

---

1 Knuth, Donald TEX and METAFONT New directions in typesetting Digital Press and the American Mathematical Society, 1979

INDEX

adapt_margins_to_grid_size, 32
adapt_to_double_bytes_format, 78
add_grey, 83
add_list, 71
add_pattern, 67
adjust_box_size, 68
after:create, 28, 31, 36, 39, 54, 57, 76
almost_prime, 13, 17
AS_EDIT_PATTERN_WINDOW, 72
AS_FONT_DISPLAY, 22
as_font_display, 21
AS_FONT_FILE, 8
as_general_font, 11, 13
ask_confirmation, 88
ask_for, 87 to 88, 99
ask_for_another_name, 103
ask_for_font_name, 103
ask_for_list, 87 to 88, 99
automatic_registers, 98

base, 9
base line, 9
before:draw, 27, 31
bias, 4
bit_pattern, 6
black_menu, 29
black_menu_order, 96 to 97
black_record, 75
black_selected, 94 to 95
black_string, 74
black_window, 26
blacken_dot, 76
box_base, 64
box_height, 38, 63
box_move_vector, 84
box_width, 38, 63
box_x, 63
box_y, 63
brush_color, 81
brush_menu, 29
brush_menu_order, 96 to 97
brush_shape, 81
bug, 15, 27, 93, 103

center_box, 64, 68, 75
center_pattern, 42
CHAR_MODE, 93
char_mode, 100
CHAR_MODE_DISPLAY, 25
char_mode_display, 21
CHAR_RECORD, 4
char_record_size, 9
CHAR_RECORD_WITH_PATTERN, 5
char_table, 11, 14
char_x, 5
char_y, 6
character_line, 13
check_dimensions, 71
check_grid_size, 33, 99
clear, 67, 83
clear_register_display, 47
clear_registers, 47
code, 4
code_list, 11, 14
code_selected, 94 to 95
command_menu, 25
command_menu_order, 102
compute_black_string, 79
compute_code, 52
compute_extremums, 72
compute_grey_string, 79
compute_grid_point, 78
compute_grid_string, 78
compute_max_height, 44
compute_parameters, 17
compute_position, 43
compute_size_and_position, 72
convert_to_number, 89
copy_in_font, 103
copy_record, 18
create, 5, 7, 13, 16, 23, 59, 67, 91, 101
create_font, 102
create_menus, 23
create_windows, 23

default_code, 49
deflate, 7
delete_record, 15
directory_path, 8, 90
DISPLAY_INTERFACE, 19
display_mode, 100
DISPLAY_SAMPLE_WINDOW, 42
displayed_x, 49
displayed_y, 49
do_draw_record, 39, 76
do_inverse_square, 76
draw, 23 to 24
draw_area, 74
draw_char, 51, 95
draw_char_box, 76 to 77

draw_code_box, 52
draw_contents, 46 to 47
draw_curve, 76, 80, 84
draw_display, 41, 43, 46,
  50 to 51, 54, 57, 75, 77
draw_element, 78
draw_frame, 47
draw_grid, 77
draw_line, 51, 78
draw_pattern, 47, 78
draw_point, 84 to 85
draw_record, 39, 76 to 77
draw_vertical_lines, 47

edit_pattern, 64
EDIT_PATTERN_WINDOW, 81
EDIT_RECORD, 62
EDITED_FONT, 13
EDITING_FACILITIES, 61
empty, 66
empty_string, 74
end_of_record, 92
erase, 80
error_message, 7
espacement, 14, 18, 45, 53, 56
espacement_x, 49
espacement_y, 49
exchange_black_and_grey, 83
execute-order, 101
execute_char_order, 101
execute_order, 101
execute_orders, 100
execute_pattern_order, 102

FILE SYSTEM INTERFACE, 3
file_head_size, 9
file_length, 90
fill_buffer, 88
first_column, 45
font, 22, 49
font_13, 22
FONT_DISPLAY_WINDOW, 48
font_display_window, 25
FONT_EDITOR, 104
FONT_EXECUTIVE, 100
FONT_FILE_MANIPULATOR, 89
FONT_GRAPHICS_WINDOW, 38
FONT_INFERIOR_WINDOW, 35
font_list, 90
font_list_file_path, 90
font_member, 93
FONT_MULTIPLE_SELECT_MENU, 36
FONT_MULTIPLE_SELECT_MULTI, 37
font_name, 8
FONT_SUPERIOR_WINDOW, 36
FONT_TEMPORARY_WINDOW, 87
font_x, 4
font_y, 4
format_bit_patterns, 17
format_pattern, 6 to 7
format_patterns, 17
French, 45

GENERAL_FONT, 10
generate_order, 85
get_attribute, 5, 12
get_black_record, 82
get_box_size, 64, 83
get_char_position, 6
get_character, 12
get_code, 4
get_code_list, 11
get_contents, 48
get_cut_record, 65, 68, 84
get_dot_color, 66, 71
get_edit_record, 82
get_espacement, 50, 54, 56
get_font, 50, 94
get_grey_record, 82
get_grid_size, 31, 64, 75
get_head, 9 to 10
get_height, 12
get_horizontal_position, 85
get_line, 67
get_line_list, 65
get_name, 14
get_number_of_registers, 45
get_number_string, 60
get_operation, 39, 50, 75
get_pattern, 6, 15
get_pattern_position, 6
get_position, 59
get_record, 5, 15, 41, 52, 65,
  67, 94, 96
get_register_contents, 46, 48
get_register_number, 47
get_selected_code, 50
get_selected_status, 41
get_size, 4, 11 to 12, 59
get_status, 46 to 47
get_title, 36 to 37, 59
get_vertical_position, 85
get_whole_record, 65, 68
get_whole_size_and_position, 65
get_width, 12
give_names, 24
global_menu, 29
global_menu_order, 96 to 97
goal, 104
GRAPHIC_INTERFACE, 34

grey_menu, 29
grey_menu_order, 96, 98
grey_record, 75
grey_selected, 94 to 95
grey_string, 74
grey_window, 26
grid size, 21
grid_size, 30, 63, 74
grid_string, 74

head_path, 8
height, 4, 45
HORIZONTAL_CODE_DISPLAY_WINDOW, 53
horz_code_window, 25
horz_scroll, 93
horz_scroll_menu, 25

improvements, 1
in, 12, 14
inflate, 7
initial_code, 49, 53, 56
initial_code_x, 49
initial_code_y, 49
initial_x, 49, 53, 56
initial_y, 50, 53, 56
initialize, 12, 22
initialize_menus, 23
initialize_windows, 23
INTRODUCTION, 1
inverse_dot, 76
inverse_square, 76, 78
items_list, 27 to 28, 31 to 32

kill, 23 to 24, 59, 101
Knuth, 109

left_margin, 26, 30
line_height, 9, 42
line_list, 64
load, 12, 14
load_buffer, 79
load_char_records, 13
load_cut_record, 68
load_edit_pattern, 68
load_font, 12, 91
load_font_list, 90 to 91
load_from_file, 9
load_from_slots, 5, 17
load_grid_string, 78
load_head, 10
load_into_slots, 5, 16
load_pattern, 10, 79
load_record, 67
load_record_list, 99
load_string, 80
load_whole_record, 68
loading_in_font, 103
lower_margin, 26, 30

maximum_width_and_height, 17
menu, 58
menu_list, 59
menus_list, 26, 28, 31 to 32
METAFONT, 108
min_raster_width, 7
move, 83 to 84
move_box, 65, 72, 83
move_line, 72
move_pattern, 65, 71 to 72

new_box_size, 84 to 85
new_code, 52, 55, 58
new_font, 91
next_brush_color, 84
next_brush_shape, 84
next_code, 52
number_of_char, 9
number_of_codes_displayed, 53, 56
number_of_fonts, 90
number_of_registers, 30, 45

open_area, 39
operation, 38, 74
outside_display, 79
outside_menu, 29
outside_menu_order, 103

PARAMETERS_DISPLAY_MENU, 58
parameters_display_menu, 29
parameters_list, 27 to 28, 31 to 32
parameters_menu_order, 97 to 98
path, 10
path_head, 10
path_name, 93
path_pattern, 10
PATTERN_DISPLAY_WINDOW, 40
PATTERN_MODE, 95
pattern_mode, 100
PATTERN_MODE_DISPLAY, 28
pattern_mode_display, 21
pattern_path, 8
PATTERN_REGISTERS_WINDOW, 44
pattern_string, 11
pattern_window, 30
pattern_window_order, 97 to 98
pile, 45
position, 38

previous_code, 52
PROPOSED IMPROVEMENTS, 105
protected register, 44
put_into_list, 92
put_into_slots, 16
put_into_string, 92
put_list_into_buffer, 70
put_one_element, 70
put_one_line, 69
put_string_into_table, 69
put_table_into_string, 70

raster width, 6
raster_width, 9, 18, 71
raster_width_and_max_height, 15
read_number, 89
record, 41 to 42, 64
record_list, 42
redraw_display, 55, 58
redraw_menus, 24
refresh, 27, 95
registers_window, 30
registers_window_order, 97 to 98
registrate_name, 91
registrated_name, 92
remove_list, 71
remove_register_contents, 46, 48
reset_espacement, 50
reset_line_list, 81
reshape, 23
reshape_box, 83
reshape_windows, 24
right_margin, 26

sample, 42
sample_window, 30
save, 15
save_edit_pattern, 70
save_font, 91
save_font_list, 91 to 92
save_head, 10
save_into_file, 9
save_into_protected_register, 46, 48
save_into_register, 46, 48
save_pattern, 10
screen_height, 22
screen_width, 22
scroll, 50, 52, 54 to 55, 57
select, 41
select_code, 51
selected_code, 49
set_black_record, 82
set_box_size, 64, 82 to 83
set_brush_color, 84
set_char_position, 6
set_code, 4
set_dot_color, 66, 71
set_edit_record, 82
set_espacement, 28, 50, 54, 56
set_font, 50, 94
set_global_parameters, 16
set_grey_record, 82
set_grid_size, 31, 64, 75
set_head, 9 to 10
set_limits, 40
set_margins, 28, 31
set_margins_and_positions, 32
set_number_of_registers, 45
set_operation, 39, 50, 75
set_pattern, 6
set_pattern_position, 6
set_position, 59
set_record, 15, 41, 43, 65, 94, 96
set_sample, 43
set_shape_color, 84
set_size, 5, 59
set_sizes, 32
set_title, 36 to 37, 59
set_values, 59
set_whole_record, 66
set_window_size, 64
show, 22, 24, 59, 100
space, 11
standard_line_and_base, 12
status, 41
string_width, 53, 56
superior, 22, 58, 100

table_of_contents, 45
take_from_slots, 17
TEX and METAFONT, 109
title, 36 to 37
TOP LEVEL SYSTEM, 86
total_height, 9
transfer_area, 40
try_to_save, 48

upper_margin, 26, 30

vert_code_window, 25
vert_scroll, 93
vert_scroll_menu, 25
VERTICAL_CODE_DISPLAY_WINDOW, 55

wait_for_vector, 99
whiten_dot, 76
width, 4
window_height, 63

window_width, 63
window_x, 38
window_y, 38
windows_list, 26, 28, 31 to 32