0055

COLLABORATORS

	TITLE : 0055	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		August 8, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

0055

1.1 Personal Fonts Maker - 2. Personal Fonts Maker Basics

2. Personal Fonts Maker Basics

This chapter is an introduction to all general concepts which are part of the Personal Fonts Maker environment. The first sections give a general overview of some important notions and terms used in typography. Many solutions which are presented in the following chapters (often coming from very different fields) should be kept in mind when working with the Personal Fonts Maker. The Font Format Description Language (FFDL) and other unique program features are also described in this chapter. Chapters 3 to 12 give more detailed instructions on how to use all the commands and functions, while chapter 13 contains several real-use examples.

2.1 From Gutenberg's Type to Computer Fonts

Writing is one of the most ingenious inventions of mankind. It allows experience to be transmitted through generations, giving speech a solid and visible aspect. It took thousands of years for writing to develop from pictograms through ideograms to the phonetic character systems now used almost all over the world. In the 15th century Johannes Gensfleisch zur Laden "Gutenberg" started a revolution which has not yet come to an end. Gutenberg invented and first used movable type to print his books.

The earliest metal type was cast by Gutenberg in 1434-44. The first typefaces were copies of the hand-written characters used in manuscripts. Type originally was a piece of hand-cast metal (composed of lead, tin and antimony, with the addition of bismuth) with a relief on one end.

It is not difficult to describe type when it is something which can be touched with the hand. A film-strip, called font, which contains negative images of the characters, is used in phototypesetting. Since electronic text processing and non-impact (e.g. laser) printing methods were introduced, the word type has become even more abstract, because it refers only to the resulting typographic image, and not to the image source itself.

Generally speaking, a font is the collection of all the various characters and symbols of a particular type design in a particular size.

Fonts may consist of very few special signs, the basic alphabet or even several hundreds of characters. In this guide, and in the terminology generally used for computer printing, a font is a set of bitmaps that constitutes an alphabet of characters of a certain point size. Concepts like bitmap and point size are explained in the following paragraphs. The words typeface and type are often used as synonyms of font, though in fact a font is a typeface in a particular type size, whereas a typeface can include fonts of different sizes.

The last five centuries have brought a series of major technical advances in the mechanization of typesetting. Every day personal computers play a more important role among the heirs of Gutenberg. Today, we use different techniques to edit, store and print computer fonts. The most important methods are based either on vector graphics or on bitmaps. With vector graphics, each character in a font is represented as a mathematical description of its shape. Outline fonts and stroke fonts are examples of fonts adopting this technique. The other approach uses arrays of dots to represent the characters in a font as bitmaps. The Personal Fonts Maker works with the second technique.

A device which produces an image consisting of separated dots is called a raster device. Conventionally, dots refer to printed output, and pixels (from PICture ELements) to screen output. In terms of raster imaging there is no difference between the two. In this guide, the word dot is generally used to focus on the output of a printer, while pixel is used where the attention is concentrated on display technology.

The Personal Fonts Maker stores the information regarding each character in a font in the form of a digital bitmap that defines the shape as a placement of dots. This method is implemented by the Amiga computer and most printers and video output systems (raster devices). The Amiga operating system (version 2.0 and beyond) can also store and scale vector fonts. Utilities like "Fountain" were designed to provide additional support for these fonts. The Personal Fonts Maker can load Amiga vector fonts, scaling them to different bitmap sizes.

Fonts based on vector graphics can occupy less memory than their bitmap-based equivalent and can more easily be scaled to another size, but are much more expensive in terms of software size and processing speed. In practice, bitmap fonts offer many more advantages to the users of personal computers like the Amiga. The Personal Fonts Maker is a powerful tool which can manipulate video fonts and convert them into printer downloadable fonts. The Amiga screen fonts and the fonts which can be printed on standard 24-pin printers are relatively small (those of 9-pin printers even smaller). Fonts in these sizes can be more compact if they are stored as bitmaps, rather than vectors. The final output quality of these fonts is much better if the characters are designed and stored as bitmaps. This is caused by the fact that even fonts based on vector graphics have to be converted to bitmaps when they are output to paper or on a screen. The result of this automatic conversion is as poor on the devices described above as it is excellent on very high resolution output devices.

Bit-mapped fonts are much easier to design and edit than their vector graphics based counterpart. With the Personal Fonts Maker, each pixel of a font is always displayed and can be modified (a so-called WYSIWYG display, meaning "What You See Is What You Get"), while users of a font editor

based on vector graphics generally need a lot of imagination to guess what the font will look like on paper.

A bitmap can be compared with a piece of graph paper in which some squares are filled in, while others are left blank. The pattern of squares corresponds to the pattern of dots that make up each character in a font. A font stored as a bitmap consists of a sequence of bits (see section 1.3.1). For each position (square) in the map there is a bit. A bit set to 1 means that there should be a dot, while a 0 indicates that the position should be left blank.

What if the Amiga fonts or the other available fonts do not have all the necessary characters? Or if an Amiga screen font cannot be used for fast and high resolution prints on a 24-pin printer? Or if a company logo is to be included in a font or a whole new font needs to be designed? Or if the shape of a letter in a font used every day is simply horrible? The Cloanto Personal Fonts Maker can solve these problems, and many more. The Personal Fonts Maker is a professional tool which was designed to let Amiga users load, edit, store and download fonts. Its main area of usefulness is as a sophisticated programmable-format font generating system that can be used in combination with most available word processors and printers. Available typefaces can be used as a point of departure to create new or partially modified character sets. The different input formats allow the user to load Personal Fonts Maker files, Amiga fonts and IFF-ILBM (which stands for "Interchange File Format - InterLeaved BitMap") brushes. It is even possible to scan a typeface from a book of the early XVI century and use the Personal Fonts Maker to download the resulting font to the printer. This was actualy done to create the "Perseus" font on the data disk. A professional scanning device was used, and the single characters were cut from the resulting picture with an Amiga paint program.

One of the most interesting features of the Personal Fonts Maker is its ability to output fonts in a user-programmable format. This means that it will be used to do things even the designers of the Personal Fonts Maker did not think of. The Personal Fonts Maker can create font files for other computers or other programs. Most important, it can create fonts which can be downloaded to a printer (sections 2.4, 2.5, 2.7, 7.3, 12.2, 12.6, 13.4 and 13.5). This has always been a problem, as almost all modern dot matrix printers accept downloaded fonts, but most printers require a different data format. Downloaded fonts have a reputation for being excessively difficult to use. That reputation is thoroughly deserved, but the Cloanto Personal Fonts Maker makes the reality different from the reputation. With the Personal Fonts Maker, users of the Amiga computers do not need to print texts in slow and poor quality graphic mode any more.

Some sophisticated off-the-shelf word processing and desktop publishing software packages for the Amiga computer offer more page layout, graphics, and character-design options than are generally available in commonly used phototypesetting systems (at a fraction of their price). However, they usually provide these capabilities with coarse character definition and/or

slow output speeds. Downloaded fonts are printed with the same speed and quality as the built-in fonts that come with the printer.

Fonts are generally used to print text on paper. The Personal Fonts Maker solves the major problem which prevented users from downloading fonts to their printer. For these reasons, this guide and the software package itself give full support to users who work with printers (without being limited to this category). The Personal Fonts Maker can send data directly to the printer. The Printer Driver Modifier program and other utilities are enclosed with the Cloanto Personal Fonts Maker package. This manual contains several useful descriptions and explanations for the users who want to download and print their fonts.

2.2 A Note on Style

The use of appropriate typefaces not only helps to communicate, but also adds an element of prestige and a personal stamp to printed material. Whether the desired look is unusual or elegant, a personal or corporate image can only be enhanced by good typography. Getting and maintaining the reader's attention is of great importance for effective communication, as only the most visually appealing messages will be read. Whether it is for printed material or for display on a computer screen, the correct use of fonts helps to catch and hold the reader's eye.

Typesetting machines and typewriters succeeded Gutenberg's press. Millions of people use typewriters to write their texts. Personal computers are quickly replacing the typewriter, though in most cases people use personal computers and printers as the equivalent of fast typewriters. Even if some early typewriters actually had printers' type on the typebars, typewriters do not create type. Modern printers make it possible to use proportionally spaced characters for greater legibility and elegance, and there is no reason to be restricted to typewriter-like fonts. Some qualities make typeset characters unique and immediately identifiable. A list of these attributes includes: proportional spacing, clean character edges, variations in character strokes (thick and thin), variety of typeface designs, design variations of a single style (bold, italic, outline, etc.). Users of the Personal Fonts Maker can create type meeting these criteria.

Usually, the choice of a typeface is from a list, and the selection is based on criteria related to the purpose of the message (utilitarian, glamour, business), the environment using it (home, office, shop), the needs of the user (space factors, sizes, typefaces), the size and needs of the audience (short-lived information, archival storage, entertainment), and the methods of reproducing the message (copier, page printer, conventional printing).

Some type has serifs (little tails on the letters, such as Times), and some is sans serif (without tails, e.g. Helvetica). Some people have a strong preference for one over the other. A sans serif type is generally used for headlines, and a serif type for the body copy.

If the reader of a long text becomes aware of the letter design, that typeface should not be used. The selected font should be transparent to the reader, rather than distracting the attention from the content of the text. A good typeface presents an even texture on the page: there are no

darker or lighter clusters of letters. Most professional designers use few decorative typefaces. "Fancy fonts" should be reserved for specialized uses, and even then, never for longer paragraphs of text.

Different type styles, like italics and boldfaces, can be used for emphasis. CAPITALS CAN ALSO BE USED FOR EMPHASIS, BUT ARE MORE DIFFICULT TO READ, because they lack the additional visual information provided by ascenders and descenders (section
2.3
). As for typefaces, the use of each different style variation should be coherent and adhere to an accepted rule of style. An excess in different type styles can draw away the reader's attention from the contextual importance of the "emphasized" text.

The Personal Fonts Maker is a professional tool, and therefore must be used with professional care. In unskilled hands, a normal, healthy-looking typeface can become horrible arabesque, which in most cases is not what the user wanted. It is extremely easy to design typefaces that are too "designy". A very limited number of typefaces is used in the major newspapers and magazines. The mixing of an inordinate number of type faces and styles is a sure sign of amateurism. Many beginning desktop publishers go crazy using every possible face and style variation in their publications. Typefaces can reflect a good deal about a publication; depending on how they are used, they can enhance the publication's image or detract from it.

2.3 Typeface Classification and Typographic Basics

Fonts either have or do not have serifs. Serifs are ending strokes on the arms, stems, and tails of the characters in a typeface. A typeface with serifs is called a roman typeface, while a typeface without serifs is called sans serif. Italic variations of a typeface are those where all characters in the font (serif or sans serif) slant to the right. Bold typefaces have thicker lines, while light typefaces have thinner and more delicate lines.

An outlined character has all its external and internal borders drawn, but is empty inside the borders. A shadowed character has a visible shadow. A reverse, or negative character has the foreground (e.g. black) and background (e.g. paper white) colours reversed. Superscripts and subscripts are the small characters which can be printed just above or below the other characters in a line. Superscripts, for example, are used to write the small numbers which appear after a word referenced by a footnote. All these variations, and many others, can be used to create a new typeface.

A type family includes the different sizes of a basic font and all its stylistically related variations. With the Personal Fonts Maker, any one of the available typefaces can be used as a point of departure to create an entire family with italic, bold, outline, shadow and other variations. Type is classified according to families. A family can contain several typefaces (e.g. bold and italic). A typeface in a particular size is called a font. A font is a typeface in a particular size.

The differences in the weight of the strokes which compose each

character vary from font to font. A text is more readable if it is printed with a roman font that has very little difference between thin and thick strokes.

It is extremely easy to calculate the number of characters that will fit in a line when all the characters which are used have identical widths, such as in typewriting. A standard typewriter can print either 10 or 12 characters per inch (abbreviated cpi). The number of characters that fit in an inch (the pitch) is a value that can be associated with the width of fixed-spaced, constant-width characters (opposed to proportionally spaced characters). A pitch of 10 cpi is called pica, while elite is used to indicate a pitch of 12 cpi. Fine, or condensed text usually has 15 or 17 cpi, while there are about 5 characters to the inch of large (enlarged) text. The Amiga system software and most other programs use this same terminology.

When different fonts and characters of various widths occupy a line, calculating the number of characters per line is a much more difficult task. An accepted standard to measure the width of the characters is based on the em measurement. For this purpose, the box containing the largest letter of the font (usually the 'M') is divided into 18 vertical slices, called relative units (RU). On more sophisticated typesetting machines, the em space can be divided into a higher number of units. A space of 6 relative units (about the size of an 'i' character, generally one third of an em-space) usually separates adjacent words in a line. Other units relative to the size of particular characters in the font are also in use. These include: the en space and the three-to-the-em space, the line (equivalent to 120 percent of the size of an em) and the figure, which is the width of a zero.

In many proportionally spaced fonts built into printers all the 10 number characters ('0' to '9') have fixed width (the figure width). This allows the numbers to be properly ordered in columns without having to change font. In some fonts the space character is equal to the average width of the characters in a line. In this way, texts (e.g. an address on a right margin of a business letter) can be more easily justified by hand on a video display which uses fixed-width characters.

When characters are processed electronically, as in the Personal Fonts Maker or in the Amiga environment, the word width may have a slightly different meaning. The width of a character (for most Indo-European alphabets) is the distance between the beginning of a character and the beginning of the following character, rather than the size of an isolated character. This means that the width of a character can include some white space before and after the character.

In typography, the size of the letters is measured in points. 72 points make up an inch. The size is determined by measuring the distance from the uppermost limit of an ascender (upward reaching letter) to the lowermost limit of a descender (downward projecting letter). Text type is usually between 9 and 12 points. A size which is often used because it offers a good compromise between space efficiency and readability is 11 points. When raster devices are used, the size is more commonly measured in pixels and dots.

Four typographic lines of reference help the designer of a font to maintain a coherent relationship from one character to another. The

Personal Fonts Maker allows the user to place four reference points (marked by little arrows), which always show the position of the lines, at the left of the character editing box (section 3.21, "Reference Points"). All characters rest on the baseline. The mean line, just above the baseline, is the upper limit of the main part of the lowercase characters. Lowercase characters that ascend upward over the mean line (’b’, ’d’, ’f’, ’h’, ’k’, ’l’, ’t’) are called ascenders. The uppermost point of an ascender is the ascender line, also called the cap line. Lowercase characters that descend below the baseline (’g’, ’j’, ’p’, ’q’, ’y’) are called descenders. The descender line limits the lowermost point of the descenders. The Personal Fonts Maker associates a default usage to each of its reference points. This is explained in section 3.21 ("Reference Points").

The distance between the baseline and the mean line is called the x-height, because it is the height of the lowercase ’x’ letter. Typefaces with larger x-heights are usually easier to read. Small capitals are capital letters at x-height.

Some notions described in this section can take on different meanings when fonts are used for video display or downloading on a printer, rather than more traditional typographic techniques. The character width and height can, for example, be measured in display pixels, screen lines or rows of printer pins.

2.4 Storage of Fonts

As already described in sections 1.3.1 and 2.1, the Personal Fonts Maker stores fonts in bitmaps. Each character is treated as a graphic image of given height and width, expressed in dots. Each dot of the image occupies one bit of memory. One bit is sufficient to store the status of the dot, which can only be "black" or "white", supposing that the font is used to print black characters on white background.

The memory occupied by a single character in the font can be calculated by multiplying the height of the character by the width of the character (expressed in dots, or bits). The result is the number of bits which are necessary to store the character data. The number of bytes occupied by the character can be calculated by dividing the result by 8.

The exact formula which can be followed to calculate the total RAM needed by the Personal Fonts Maker to store a font’s graphical data is:

```
TotalBytes = 257 * (((XMAX + 7) >> 3) * YMAX)
```

Sections 7.3.1 and 7.3.2 explain the "X Max" and "Y Max" parameters. The ’>>’ (shift right) operator is described in section 2.7.3.3 ("FFDL Bit Manipulation Operators").

The graphic data of a font containing 127 characters in a 36 by 24 matrix will, for example, occupy a minimum of 13716 bytes (36 x 24 x 127 / 8). This is a typical size for a letter quality font downloadable on a

24-pin printer. In practice, the number of bytes can be different, as proportional characters do not have all the same width, and the number of downloaded characters can be optimized to include only the characters to be printed. Depending on the purpose for which the font is stored, some additional bytes can be added to round the number of bits in a row to a multiple of 8 or 16, or give more detailed information on how the font is to be represented. In the previous example, the number 36 should have been rounded to the next multiple of 8, which is 40, to calculate the amount of memory occupied internally by the program. Section 1.3.1 ("Measurement Units") contains a similar example.

The Personal Fonts Maker can store and load fonts in two predefined formats (PFM and Amiga), and generate font files in any format programmed by the user, or selected from one of the predefined font description files that come with the software package. Single characters, or parts thereof, can be loaded and saved as standard Amiga IFF ILBM graphical files.

The Personal Fonts Maker stores fonts in its own format by default. PFM font files follow the IFF (Interchange File Format) standard. Font files in this format contain more information (e.g. X/Y aspect ratio, reference points and other attributes) than the Amiga font files, in almost half the space. A single PFM IFF file can contain font data, a character set description, a colour palette and more. Appendix O contains a very detailed technical description of the data structures used by the Personal Fonts Maker to store its fonts. Additional information on the IFF format can be found in the official Commodore documentation. Sections 4.3 and 4.4 explain how to load and save fonts in this format.

Amiga fonts can be loaded with the Personal Fonts Maker. They can be modified and saved in the Amiga or Personal Fonts Maker formats. Most important, Amiga fonts can be downloaded to a printer. In this way, it is possible to print a text using the same font displayed on the screen. A font of height 24 can be downloaded to the printer, while a font of the same family, but of smaller height (the resolution of the screen is usually not as high as that of the printer) can be used for true WYSIWYG screen display. All fonts which can be displayed with the Personal Fonts Maker can be stored in Amiga format. Since this is not the default PFM format, and font files in this format contain less information than standard PFM font files, the verbs import and export are often preferred to describe load and save operations in Amiga format. Fonts saved in Amiga format can be loaded by all Amiga software using standard fonts. Sections 4.5 and 4.6 explain how to load and save fonts in the Amiga screen font format.

One of the most interesting features of the Personal Fonts Maker is its capability of creating font files in a user programmable format. With the powerful FFDL (Font Format Description Language), fonts can be downloaded to any printer, or imported by other programs on different computers. It is even possible (but not necessary, as there is a special function described in section 7.8 to do so) to create a standard Workbench ".info" icon file using the FFDL. FFDL descriptions can be written with a text editor or with the requester described in section 7.3. The PFM disks contain several FFDL descriptions ready to be used to download fonts to some of the most used printers. More detailed information regarding this feature can be found in sections

2.1

,

2.7
, 7.3, 12.2, 12.6, 13.4 and
13.6. Many examples in this guide describe FFDL applications for printers. This is because printer download formats are among the most complex formats to handle, and most Amiga users feel the need of fast and high quality prints with custom fonts. This of course does not limit the use of the FFDL, which is also used in several other fields, ranging from the video sector to the creation of fonts for other computers.

Graphic data in IFF ILBM (Interchange File Format - InterLeaved BitMap) formats can be used to load and save a PFM brush (section 5.1). With this function, graphic data can be exchanged with popular paint programs. This is particularly useful when fonts are to be extracted from a large picture, as an image generated by a scanning device (also see section

2.1
) or a video camera.

It should be noted that the IFF ILBM format is different from IFF CPFM, used by the Personal Fonts Maker to store its data. Both files adhere to the Interchange File Format standard, but ILBM files are used for bitmap graphics, while CPFM (Cloanto Personal Fonts Maker) files contain Personal Fonts Maker font and character set data.

2.5 Downloaded Printer Fonts

All devices which do not print characters as a single entity (like daisy-wheel printers) build their symbols from a bitmap. These printers may adopt different techniques, like matrix impact, ink jet, optical or thermal transfer. The Personal Fonts Maker can be used to download fonts to almost all these different types of printers which accept downloaded fonts.

There are some technical details regarding the printer which should be known before a font is designed and downloaded. The following paragraphs will explain how the printer’s memory works, how to transfer the font data to it, and how each dot is finally put on paper.

Printers are manifactured with a limited number of fonts which are stored in ROM (Read Only Memory). It would not be possible to store enough typefaces in different sizes to match all user’s needs in ROM. For this reason, modern printers support the downloading (installation) of fonts from the computer. Downloaded fonts are often called soft font, as opposed to the "hard" fonts built into the printer. These fonts can be designed and edited with the Personal Fonts Maker, saved on disk, and downloaded to the printer as required.

The printers which support the downloading of fonts have a built-in RAM. If no font is downloaded, all of this memory can be used as a buffer to store the data received by the computer which has not yet been printed. In most cases the printer’s memory can be shared by the buffer and the download functions.

The content of the RAM is cleared whenever the printer is switched off. This means that the downloaded soft fonts will also disappear from the printer’s memory. There are different ways to make sure the preferred soft

fonts are always in the printer's RAM. Some programs, like the Personal Write word processor, are clever enough to automatically download any soft font selected by the user. "PrintRawFiles" is a utility (described in section 12.2) included with the Personal Fonts Maker package which can be used to download a file to the printer. The Personal Fonts Maker itself can, of course, be used to download a font.

Most printers have special commands which let a ROM-font be copied to the printer's RAM. This can be useful if the Personal Fonts Maker is used to modify only a few of the characters which already exist in the printer's ROM. Once a ROM font is copied to the RAM, it can be partially overwritten by user-designed characters.

With some printers it is not possible to download and use more than one font at a time, while others can handle two or more fonts. The RAM of some printers is not large enough to support the downloading of a full high resolution font. In such a case, if no additional memory can be added, the characters which are not used in the text to be printed should not be downloaded. This saves a lot of memory. The "TextChars" utility, described in section 12.1, automatically analyzes the contents of an ASCII text file and creates a PFM macro which excludes all characters which are not contained in the file.

Characters having a code greater than 127 cannot be downloaded on some printers. Some Amiga printer drivers also have problems handling these characters. The Printer Driver Modifier (described in chapters 9 to 11), which is included with the Personal Fonts Maker package, can be used to enable the download and printing of even these "forbidden" characters.

Downloaded fonts can be printed with the same speed and quality as the fonts built into the printer. Unfortunately, downloading a font is not an easy task. Amiga screen fonts cannot just be "sent" to the printer, nor should they be printed in slow and poor-quality graphics modes. Printers require the use of special control codes and initialization sequences. The bitmaps must be formatted so that they can be used by the printer's graphic interpreter as a series of printable characters. The reason for which the download functions are in most cases unused, lies in the fact that there is no standard way to download a font. A program which is to download fonts to different printers should be able to use a different format for every printer on the market. This is exactly what the Personal Fonts Maker does. Section 2.7 explains the use of the Cloanto FFDL (Font Format Description Language), which is the key which opens the doors to all printers.

The Personal Fonts Maker can create soft fonts in any format defined through a sequence of FFDL commands. The resulting data can either be directly downloaded into the printer's memory, or be saved to a file from where it can be used whenever it is needed (see section 4.13 for more details). Please note that a data file saved in a printer download format cannot be loaded and edited by the Personal Fonts Maker. For this reason, a user-designed font should always be saved either in Amiga or PFM format, even if a printer file has already been written.

The hardware used to print the bitmap can limit the accuracy, frequency

and size of the dots. It is important to understand how a printer puts the downloaded bitmaps on paper, since this can have implications on the early design stages of a font. The shape of the dots is usually quite different from the rectangular pixels displayed on the screen. Printer pins are normally round, but can also be more or less elliptical. Thermal transfer printers print square or rectangular dots.

To make diagonal lines of circular dots more readable, some impact matrix printers overlap the dots. Ideally, good print quality can be achieved through a dense matrix of small dots, as expensive laser printers use. Impact printers are limited by the same technology which makes them so inexpensive and popular. The pins in a printer head cannot be too small, or they would pass through the ribbon. The overlapping of dots is a good compromise to achieve a better print quality. Horizontal overlapping can be achieved by passing the print head more than once on the same line. The paper can be shifted up by a fraction of the dot diameter after a first print pass, so that dots can be overlapped vertically. Some printers have more rows of pins on the head, so that they can print overlapping dots in one pass.

Some impact matrix printers cannot print two horizontally adjacent dots in one pass of the printer head. This is caused by the fact that the electromechanical device which presses the inked ribbon to print a dot on the paper needs some time before it is ready for another "fire". For this reason, when the printer is in letter quality mode, it passes on the same line several times. In high speed draft mode, it simply does not print adjacent dots. Special draft mode fonts are designed in such a way that there are no adjacent dots in the bitmap (e.g. the "Oberon_Draft_24" font).

2.6 Program and Font Parameters

Several program and font parameters can be set to adapt the Personal Fonts Maker to the most different user needs and tastes. Different parameters are used to store the level of the audio volume, the screen colours, the appearance of the font on the screen and on paper, and many other details.

All parameters can be set through the Amiga user interface, usually by selecting the proper menu item in the "Preferences" menu (chapter 7). The settings can be stored and loaded again at any time, using the functions described in sections 7.1 and 7.2. Using the "Save Preferences" and "Load Preferences" functions, the user does not need to regulate the parameters every time the program is used. Once the parameters are stored, they can be loaded automatically or manually every time the Personal Fonts Maker is used.

All parameters have a default value, programmed by the designers of the Personal Fonts Maker. The default value is automatically set until another value is specified.

The parameters can be logically grouped in two classes: font parameters and program parameters. Font parameters only affect the way fonts are represented on screen, on paper or as described through the FFDL (section

). Most font parameters can be accessed through the FFDL. Program parameters, on the contrary, allow the user to control the user interface of the Personal Fonts Maker.

The Personal Fonts Maker allows the user to work with two different environments at the same time. Two completely different fonts, font parameters and character sets can coexist in one work session. The user can switch between the two environments with the "Font" gadget, described in section 3.2. When the user works with font number one, the first set of font parameters is applied to the font. When the user switches to font number two another font can be edited, and the second set of font parameters will be used. Since the program parameters are shared by the two font environments, the user interface will still use the same program parameters.

As explained in section 1.11, the "StartupF1.prf" and "StartupF2.prf" files in the "PFM:PFM_Prefs" drawer contain the initial settings for the two environments. The Personal Fonts Maker automatically tries to load these two parameter files when the program is started.

The Personal Fonts Maker stores the parameters as plain ASCII (pronounced "asky", from "American Standard Code for Information Interchange") text files. This means that the parameters contained in the files can be accessed and modified either through the user interface of the Personal Fonts Maker, or with a text editor or a word processor like Personal Write. Users who are not interested in accessing the parameter files directly may skip the remaining part of this section.

A parameter file is a plain ASCII text file. The parameter file must begin with the header "PFM PREF" (from "Personal Fonts Maker PREFerences") to be recognized as such by the Personal Fonts Maker. A four letter abbreviation is associated with each parameter of the Personal Fonts Maker. Exactly as menus and gadgets can be used to modify a parameter with the standard Amiga user interface, the four-letter codes allow the user to set the parameter from within a text file. The four letters should be written in upper case, to be coherent with the style of the FFDL commands, though the Personal Fonts Maker will also interpret lower case characters.

A parameter file consists of the header ("PFM PREF"), followed by one or more parameter settings. To assign a given value to a parameter, the four letter code (keyword) must be followed by an equal sign (’=’) and the desired value. The macro file name keyword ("MCRO") requires an additional parameter (the letter associated with the macro referred to) before the equal sign. The RNGE parameter requires two values (separated by one or more spaces or TABs) to follow the equal sign.

Numerical parameters can only be followed by digits. File name string parameters are alphanumerical parameters, i.e. they can be followed by a sequence of letters, numbers and other signs, preceded and followed by a quote character (’"’, as in "string"). No quote characters may appear within the string. File name parameters are used to select the files to be loaded. The third type of parameter is the FFDL sequence. An FFDL sequence must not be enclosed by quotes. Space and TAB characters before an FFDL sequence are optional, as for all other parameters. Section
2.7
explains

how to write an FFDL sequence.

The parameter (all three types) and the value assigned to it must stay on the same line. The assignment of a parameter (numerical or alphanumerical) can be followed by another parameter or by a comment. A comment must begin with a semicolon (';') and cannot be followed by other data (which would be interpreted as being part of the comment) on the same line. FFDL sequences cannot be followed by other data nor by a comment, as any additional characters on the same line would be interpreted as being part of the FFDL sequence.

The maximum length of a line in a PFM parameter file is 400 characters. Some text editors cannot handle such long lines. In these cases, a word processor like Personal Write can be used to edit and save the text. The word processor must be able to save the text as an ASCII file.

Space (' ') and TAB (decimal ASCII code 9) characters can be used to separate equal signs, semicolons or other elements in each line. Space and TAB characters are skipped by the Personal Fonts Maker (i.e. the program behaves as if they did not exist), unless they are part of a string value assigned to a file name parameter.

If a parameter file containing syntax errors, or incoherent data is read, all parameters in the file are ignored and the previous values (the default values, if the initial settings were being loaded) are used. An error message is displayed unless the error occurs during the initial load of the default settings ("StartupF1.prf" and "StartupF2.prf" files).

The following is an example of a valid preference file:

```
PFM PREF

WBEN = 0            ;comment: this line closes the Workbench screen
COOR = 2

ICON = 2  GRID = 1  ;other data can follow the assignment of a parameter

MCRO A = "PFM:PFM_Macros/MyMacro.mcr"  MCRO B = "PFM:PFM_Macros/Out.mcr"

SQON = ESC \& (0) CNUM CNUM (1) XSIZ (1) VDAT   ITAF=2
         ;WARNING: the above line will cause a syntax error message
         ;         to be displayed during the execution of the FFDL
         ;         sequence. ITAF will not be set to 2

SQON = ESC \& (0) CNUM CNUM (1) XSIZ (1) VDAT
ITAF = 2              ;this is what we wanted for SQON and ITAF
```

The following sections list all parameters which can be accessed through four-letter codes in a parameter file. Some of these parameters (the font parameters) can also be accessed through the FFDL, described in section

2.7

. Appendix N contains a table of all parameters.

2.6.1 The AUDC Parameter

Parameter Type : Numerical
Valid Range   : 0 to 64
Default   : 64
Example   : AUDC = 30
More in Sections: 7.13

This parameter controls the volume of the standard feedback sounds of the Personal Fonts Maker.

2.6.2 The AUDE Parameter

Parameter Type : Numerical
Valid Range   : 0 to 64
Default   : 64
Example   : AUDE = 64
More in Sections: 7.13

This parameter controls the volume of the acoustic error signals of the Personal Fonts Maker.

2.6.3 The BRHA Parameter

Parameter Type : Numerical
Valid Values   : 0, 1, 2, 3, 4
Default   : 2
Example   : BRHA = 1
More in Sections: 5.9, 6.6.2

The position of the brush handle is determined by this parameter. A value of 0 places the handle at the upper left of the brush, 1 at the upper right, 2 at the centre, 3 at the lower left and 4 at the lower right of the brush.

2.6.4 The COLR Parameter

Parameter Type : Numerical
Valid Range   : -15 to +15
Default   : 0
Example   : COLR = 0
More in Sections: 7.14

This parameter controls the level of red in the user interface colours, relative to the default setting.

2.6.5 The COLG Parameter

Parameter Type : Numerical
Valid Range   : -15 to +15
Default   : 0
Example   : COLG = 5
More in Sections: 7.14

This parameter controls the level of green in the user interface colours, relative to the default setting.

2.6.6 The COLB Parameter

```
Parameter Type  : Numerical
Valid Range     : -15 to +15
Default         : 0
Example         : COLR = -6
More in Sections: 7.14
```

This parameter controls the level of blue in the user interface colours, relative to the default setting.

2.6.7 The COOR Parameter

```
Parameter Type  : Numerical
Valid Values    : 0, 1, 2
Default         : 0
Example         : COOR = 1
More in Sections: 7.5
```

A value of 0 causes no coordinates to be displayed on the title bar. If the value is 1 or 2, the coordinates will be displayed with a starting position of 0:0 and 1:1, respectively.

2.6.8 The CSET Parameter

```
Parameter Type  : String (File Name)
Maximum Length  : 192 Characters
Default         : "PFM:PFM_CharSets/PC/PC_Usa2.set"
Example         : CSET = "PFM:PFM_CharSets/PC/PC_Usa2.set"
More in Sections: 4.8
```

This string must contain the entire path and file name which the program can use to access a file containing character set data. The maximum length of the path is 128 characters.

2.6.9 The EPIL Parameter

```
Parameter Type  : FFDL Sequence
Maximum Length  : 400 Characters
Default         : no sequence
Example         : EPIL = ESC \% (1)
More in Sections:
                2.7
                , 7.3
```

This FFDL sequence describes the format of the final data (epilogue) of an FFDL output.

2.6.10 The FILR Parameter

```
Parameter Type  : Numerical (four bits)
Valid Values    : 0 to 15
Default         : 13
Example         : FILR = 1
More in Sections: 7.11
```

This parameter consists of four bits. The first bit determines whether paths in the file requester are to be expanded (section 7.11.1). The second bit determines whether icon (".icon" and ".info") file names are to be displayed in the file requester, while the status of the third bit controls whether a double-click is handled in the file requester (section 3.23) and macro requester (section 6.1). If the fourth bit is set, the Personal Fonts Maker displays a warning message before overwriting data on an existing file. The desired value can be obtained by adding 1, 2, 4 and 8 if the first, second, third and fourth bit are set, respectively. There are sixteen possible combinations: 0 = no path expansion, no icon files, no double-click, no overwrite warning; 1 = expand path names, no icon files, no double-click, no overwrite warning; 2 = no path expansion, list icon files, no double-click, no overwrite warning; 3 = expand path names and list icon files, but no double-click nor overwrite warnings. The settings 4 to 7 are the same as 0 to 3, respectively, but double clicks are interpreted. Settings 8 to 15 are similar to those from 0 to 7, only that overwrite warning messsages are activated.

2.6.11 The GRID Parameter

```
Parameter Type  : Numerical
Valid Values    : 0, 1, 2
Default         : 1
Example         : GRID = 1
More in Sections: 7.6
```

This parameter controls the appearance of the grid in the character editing box. If the parameter is set to 0, no grid is displayed. If the value is 1 or 2, the grid is displayed consisting of coloured lines or blank lines between the dots, respectively.

2.6.12 The ICON Parameter

```
Parameter Type  : Numerical
Valid Values    : 0, 1, 2
Default         : 1
Example         : ICON = 1
More in Sections: 7.8
```

A value of 0 means that no Workbench icon files will be saved along with the files saved by the Personal Fonts Maker. If the value is 1 or 2, icon files are always saved. A value of 1 will cause the standard image to be used for font files, while if the value is 2, the image used for the Workbench icon will be extracted from the currently displayed character.

2.6.13 The ITAF Parameter

```
Parameter Type  : Numerical
Valid Range     : 1 to 255
Default         : 3
Example         : ITAF = 2
More in Sections: 5.6, 6.6.8, 7.4
```

This parameter determines the number of consecutive lines which have to be slanted to the right by one dot, relative to the lines below them when the "Italicize" function is called.

2.6.14 The JOIN Parameter

```
Parameter Type  : Numerical
Valid Values    : 0, 1
Default         : 1
Example         : JOIN = 1
More in Sections: 7.9
```

The Personal Fonts Maker allows the user to work with two fonts at a time. If this parameter is set, whenever a character is selected from one font, the character having the same code in the other font becomes the current character of that font. This means that the current character of the two fonts will always have the same code, i.e. it will be the same character, if one character set is used for both fonts. If the parameter is not set, each font has its own, independent, current character.

2.6.15 The LANG Parameter

```
Parameter Type  : Numerical
Valid Range     : 0 to 255
Default         : 0
Example         : LANG = 0
More in Sections: 7.12
```

In multi-lingual versions of the Personal Fonts Maker this value controls the user interface language. The following languages, for example, are associated with the first numbers: 0 = English, 1 = Italian, 2 = German, 3 = French and 4 = Spanish. There are slight differences between the British and the American English versions of the Personal Fonts Maker texts.

2.6.16 The MCRO Parameter

```
Parameter Type  : String (File Name)
Maximum Length  : 192 Characters
Default         : no macro
Example         : MCRO A = "PFM:PFM_Macros/Outline.mcr"
More in Sections: 6.1
```

This string must contain the entire path and file name which the Personal Fonts Maker can use to access a macro file. The maximum length of the path is 128 characters. The letter before the equal sign indicates the key (from <A> to <Z>) to which the macro is to be associated.

It is possible to select up to 26 different macros (in the ASCII character set the letters from A to Z are 26) from the same parameter file.

2.6.17 The PROL Parameter

```
Parameter Type  : FFDL Sequence
Maximum Length  : 400 Characters
Default         : no sequence
Example         : PROL = ESC \x (1) ESC \: NUL NUL NUL
More in Sections:
                2.7
                , 7.3.5
```

This FFDL sequence describes the format of the initial data (prologue) of an FFDL output.

2.6.18 The RNGE Parameter

```
Parameter Type  : Numerical (two values)
Valid Range     : -1, -2, -3 and 0 to 255
Default         : 0 255
Example         : RNGE = 0 255
More in Sections: 7.3.9
```

The RNGE variable specifies the starting and ending characters of the font for which the SQON and SQOF FFDL sequences must be repeated.

2.6.19 The SQON Parameter

```
Parameter Type  : FFDL Sequence
Maximum Length  : 400 Characters
Default         : no sequence
Example         : SQON = ESC \& (0) CNUM CNUM (1) XSIZ (1) VDAT
More in Sections:
                2.7
                , 7.3.6
```

This is the FFDL sequence associated with the active characters.

2.6.20 The SQOF Parameter

```
Parameter Type  : FFDL Sequence
Maximum Length  : 400 Characters
Default         : no sequence
Example         : SQOF = (0)
More in Sections:
                2.7
                , 7.3.7
```

This is the FFDL sequence associated with the characters which are not active.

2.6.21 The STCM Parameter

```
Parameter Type  : Numerical (two bits)
Valid Values    : 0, 1, 2, 3
Default         : 3
Example         : STCM = 3
More in Sections: 3.12, 4.3, 4.5, 7.3.1, 7.3.2, 7.10
```

The first bit determines whether the maximum or the proportional stretch mode is to be used. If the second bit is set, the stretch function is automatically used when data is exchanged between fonts through the character buffer's "Paste" function. The following values are possible: 0 = maximum stretch, no stretch from buffer; 1 = proportional stretch, no stretch from buffer; 2 = maximum stretch, buffer data stretched automatically; 3 = proportional stretch, automatic buffer stretch.

2.6.22 The WBEN Parameter

```
Parameter Type  : Numerical
Valid Values    : 0, 1
Default         : 1
Example         : WBEN = 0
More in Sections: 4.15, 7.7
```

If the parameter is set to 0, the Personal Fonts Maker tries to close the Workbench screen. The default is 0 if the Personal Fonts Maker is loaded on a computer with less than 1 Mbyte of RAM.

2.6.23 The XDPI Parameter

```
Parameter Type  : Numerical
Valid Range     : 1 to 65535
Default         : 360
Example         : XDPI = 360
More in Sections:
                2.7.2.20
                , 7.3.3
```

This parameter contains the number of horizontal dots per inch of the output device (e.g. printer or video).

2.6.24 The XMAX Parameter

```
Parameter Type  : Numerical
Valid Range     : 1 to 255
Default         : 36
Example         : XMAX = 37
More in Sections: 6.6.57,
                2.7.2.21
                ,
                2.7.2.22
                , 3.4, 7.3.1
```

The maximum horizontal size of each character, expressed in dots, is defined by this parameter.

2.6.25 The YDPI Parameter

```
Parameter Type  : Numerical
Valid Range     : 1 to 65535
Default         : 180
Example         : YDPI = 360
More in Sections:
                2.7.2.24
                , 7.3.4
```

This parameter contains the number of vertical dots per inch of the output device.

2.6.26 The YMAX Parameter

```
Parameter Type  : Numerical
Valid Range     : 1 to 255
Default         : 24
Example         : YMAX = 24
More in Sections:
                2.7.2.25
                , 7.3.2
```

The maximum vertical size of the font (i.e. all characters in the font), expressed in dots, is defined by this parameter.

2.7 Programming the Output Format: The Cloanto FFDL

The Font Format Description Language is a powerful tool which can be used to describe the format of the data to be output. This is necessary whenever the Personal Fonts Maker is used to create font data (a font file, or printer download data) in a format different from the standard Amiga or Personal Fonts Maker font file format. The FFDL makes the Personal Fonts Maker so flexible that it can create download files for the most different printers (see sections
2.1
and
2.4
).

Section 7.3 explains how FFDL sequences can be combined to create font data downloadable to a printer or ready to be processed by another program or system. This section explains how to use the different "bricks" of the FFDL, in order to write an FFDL description which can be interpreted by the Personal Fonts Maker.

The following three lines are examples of FFDL sequences:

```
w(\A + 32)
ESC \x (1) ESC \: NUL NUL NUL
ESC \& (0) CNUM CNUM (1) XSIZ (1) VDAT
```

An FFDL sequence is a text string written by the user. The FFDL is used to specify what data is to be output, in which order and in what format. The FFDL allows the user to specify the data format as a sequence of data units. Data units in an FFDL sequence are interpreted in their order of appearance (left to right) to generate the associated output. Successive data units in a string are separated by one or more space characters.

A data unit can consist of constants and variables. A constant is a number which is defined by the user and is not varied by the program. A variable is a value which is referenced to by the user, but is defined by the program at output time. Variables are written as four-letter codes. Only capital letters can be used for variables. The number 27, for example, is a constant. The width of a character (XSIZ) in a font is a variable, as it is determined at output time, depending on the width of the character currently processed. Several font parameters can also be accessed as variables.

Both constants and variables can be terms of algebraical and logical operators. Operators are used to perform "calculations". The plus sign ('+') is an example of an algebrical operator. Other operators are listed in section
2.7.3
.

There are different kinds of data units. Some font parameters (like XDPI) are also variables which are immediately recognized as data units. Constants can be written either as standard ASCII control code abbreviations (like ESC), as ASCII character constants (having a decimal code in the range 33-126) identified by a '\' prefix (e.g. \A), or as a number enclosed in parentheses, like (32). Parentheses can contain constants, variables and operators. All the data between two parentheses is called expression, and is interpreted and treated as one data unit.

There are a few data units with peculiar properties. CIDT, HDAT, HICD, VDAT and VIDT cannot be treated as operands (except as arguments in LENG). Instead, they are already complete data units, which tell the Personal Fonts Maker that the character graphic data is to be output in a particular format. LENG must be followed by an expression contained between parentheses. REPT must also precede two parentheses, but the latter must contain two expressions separated by a comma (','). All variables and data units are described in detail in the following sections.

2.7.1 FFDL Constants

Constants are the most used data units and easiest to understand. A constant is a number which is written by the user and is not modified by the Personal Fonts Maker at output time.

When the FFDL is used to describe the format of font data to be downloaded to a printer, constants are typically used to make up the control sequences as described in the printer's documentation.

A subset of the FFDL, where only constants are interpreted, is used by the Printer Driver Modifier (chapters 9 to 11), as only standard

printer control sequences need to be described.

There are three ways of using constants to create a data unit. If the code to be output can be logically associated with the character in the standard character set (see section
2.8
) having the same code, then it may be most intuitive to type that character. The character must be prefixed with a backslash ('\'), so that the Personal Fonts Maker can interpret it as a character constant. Otherwise the program would search for the character in its internal dictionary of variables and constants, displaying an error message if the character cannot be found there. A data unit which is to cause the output of the capital letter 'A' (code 65) would be: \A. A backslash (code 92) would be written as: \\. Appendixes B and D list the codes which may be useful to create data units in this format. The quote sign ('"', ASCII code 34) can be used to enclose a sequence of ASCII characters. If either the quote or the backslash character are to appear in the string, they must be preceded by a backslash. For example, \A \B \C \" \\ SP \D could be written as "ABC\"\\ D".

The second way to output a constant value is to write a data unit in the form of an ASCII control code abbreviation. There are 33 standard two- or three-letter abbreviations for the codes from 0 to 32. These are:

```
00-09:   NUL  SOH  STX  ETX  EOT  ENQ  ACK  BEL  BS   HT
10-19:   LF   VT   FF   CR   SO   SI   DLE  DC1  DC2  DC3
20-29:   DC4  NAK  SYN  ETB  CAN  EM   SUB  ESC  FS   GS
30-32:   RS   US   SP
```

Appendix B has a detailed listing of these codes. The codes do not need to be prefixed by a backslash. The abbreviations must be written in capital letters.

The third method which can be used to define a data unit with constant data is to write the code in numbers between parentheses. This is not immediately recognized by the program as a constant, since the expression contained in the parentheses must first be evaluated (section
2.7.3
), but the result is the same: a constant.

Decimal, hexadecimal and octal numbers are accepted between parentheses. Hexadecimal numbers must be preceded by a "0x" prefix, while octal numbers must begin with a '0' digit. This follows the standard adopted by the C programming language, and explained in section
2.7.3
.

When the handbook of a printer says that a sequence like

```
ESC % 1
```

or

```
CHR$(27) "%" CHR$(1)
```

has to be sent to the printer (these are two ways to represent the same control codes) to activate a certain function, the equivalent Personal

Fonts Maker data units could be:

```
     (27) (37) (1)
or
     ESC \% (01)
or
     ESC (0x25) SOH
```

This is just an example to show how the same data units can be written in different ways with the FFDL.

It should be noted that a number alone (without parentheses), or a character without the preceding backslash, are not data units. When the Personal Fonts Maker encounters a number or another character without the signs just mentioned, it searches to see if it recognizes that group of characters as a variable. If it cannot find a matching variable, an error message is displayed and the output of the FFDL sequence data is aborted.

2.7.2 FFDL Variables

The Personal Fonts Maker allows the user to create a definition of the font data format as an entity which is completely independent from the font data itself. This means that once an FFDL sequence has been written, it can be used for different fonts in different sizes, without having to be rewritten. To do this, variables can be inserted in the FFDL sequences.

Variables are data units written as codes consisting of four capital letters. When the FFDL sequence is read by the program to output the data, the values currently associated with each variable are output.

Some variables are character relative. This means that different values can be associated to the variable, depending on the character currently processed. These variables are relative to the current character. The variable CNUM, for example, always contains the code associated with the current character. These variables are often described as "current character-relative".

As described in section 7.3, some FFDL sequences can be repeated for a series of characters. This means that the same FFDL sequence is used to output data of many different characters. Current character-relative variables are generally used only in these FFDL sequences. If a current character-relative variable is used in another FFDL sequence, then the current character will be the character currently selected and displayed in the font editing box.

The following sections explain each variable in detail.

2.7.2.1 FFDL Variables: ATRB

This variable contains one bit for each of the font style attributes which the Personal Fonts Maker can handle (chapter 8). The variable describes the style of the current font, and is updated whenever a font is loaded or imported, or when the style flags are modified manually, as described in chapter 8. The following table shows the meaning of each

bit.

BIT	ATTRIBUTE	SECTION
0	Italic	8.1
1	Bold	8.2
2	Light	8.3
3	Underlined	8.4
4	Outline	8.5
5	Shadow	8.6
6	Superscript	8.7
7	Subscript	8.8
8	Enlarged	8.9
9	Condensed	8.10
10	Reverse	8.11
11	Serif	8.12
12	Draft	8.13
13	Fixed Pitch	8.14
14	Right to Left	8.15
15	Landscape	8.16

For example, the following sequence outputs a 1 (one) if the "Shadow" flag is set, 0 (zero) otherwise:

```
(ATRB >> 5 & 1)
```

Similarly, this sequence outputs a 1 (one) if either the "Outline" and/or the "Shadow" flags are set, 0 otherwise:

```
((ATRB & (1 << 4 | 1 << 5)) != 0)
```

The ATRB is particularly useful for adding style information to downloaded fonts. Most page printers support similar functions.

2.7.2.2 FFDL Variables: CHQT

This variable (the name is an abbreviation of "CHaracter QuanTity") contains the number of characters in the selected range of the font which are active (see section 3.10). If no character is active, CHQT is 0.

2.7.2.3 FFDL Variables: CIDT

This data unit outputs the graphical bitmap of the current character. The vertical columns of the bitmap are processed and output from left to right, in two passes (CIDT comes from "Character Interlaced DaTa"). This format is used by some printers which - as described in section 2.5 - shift the paper up by a fraction of the dot diameter after a first print pass, so that dots can be overlapped vertically. The CIDT data unit (like VIDT) outputs data in a format which is ready to be used by these printers.

In the first pass, only dots appearing at odd lines (i.e. odd numbered positions from the top, counting from one) in each column are output. Dots

which are contained in even-numbered rows are output in the second pass.

CIDT outputs the bitmap of the current character in two single passes (character priority mode). This is different from VIDT (section

2.7.2.18

), which repeats the odd/even passes for each column in the character (column priority mode).

This format is generally used only to create high quality download fonts, as draft and high-speed fonts are usually not printed with multi-pass techniques.

This data unit cannot be used as a term of an expression. The variable contains graphic data, which cannot be handled by the operators described in section

2.7.3

. No prefix codes can be used before this variable. CIDT can, however, appear as an expression in the LENG() function. HDAT, HICD, VDAT and VIDT work in the same way. These bitmap output formats are described in sections

2.7.2.8

,

2.7.2.9

,

2.7.2.17

and

2.7.2.18

.

2.7.2.4 FFDL Variables: CNUM

This is the code of the current character. This is not the Amiga character set code of the character, which can be obtained with the variable described in section

2.7.2.5

. It is, on the contrary, the code displayed on the screen when the characters are edited (section 3.3).

2.7.2.5 FFDL Variables: EQAM

This is the code of the current character translated into the Amiga character set code. If no equivalent character was defined in the Amiga set, then EQAM (from "EQuals AMiga") will contain -1.

Sections

2.8

and 4.10 have more on character sets.

2.7.2.6 FFDL Variables: FCLM

The first column of the current character which contains at least one coloured dot is referenced by this variable. The code of the leftmost column is zero, and may also be returned to indicate that the character is

completely empty. FCLM and LCLM (section
2.7.2.12
) can be used in fixed
pitch fonts in a way similar to the KERN-SPCE combination in
proportionally spaced fonts.

2.7.2.7 FFDL Variables: FRST

This variable indicates the code of the first character on which the SQON and SQOF FFDL sequences are being repeated. This may not be the same as the first value of the RNGE parameter (section
2.6.18
) if one of the
special codes (-1, -2 and -3) is used to define the range, as described in section 7.3.9. FRST is not relative to the character currently processed, but has the same value throughout the whole output cycle. If the start code of the range is defined by a -1, -2 or -3, FRST respectively contains the code of the first "On" character, the first "Off" character or the character which was the current character before the data output began.

FRST is generally used to describe an "optimized" font download format (section 12.6.4 , "Parameter Files: Printer Font Descriptions"), as in the following example:

```
; Fujitsu Optim
; copy ROM set to RAM,
; only one command to activate the redefinition of
; all characters from FRST to LAST

PROL = ESC \: NUL NUL NUL ESC \& (0x10) FRST LAST

; XSIZ (> 0) means that (XSIZ*3) bytes of data follow in VDAT
SQON = XSIZ VDAT

; (0) means that no data is sent ("Off" character)
SQOF = (0)

; activate the downloaded characters
EPIL = ESC \% (5) (0)
```

2.7.2.8 FFDL Variables: HDAT

This data unit causes the output of the graphical bitmap of the current character. The horizontal lines of the bitmap are processed and output from top to bottom, as horizontal rows of dots (HDAT comes from "Horizontal DATa").

This data unit cannot be used as a term of an expression. The variable contains graphic data, which cannot be handled by the operators described in section
2.7.3
. No prefix codes can be used before this variable. HDAT can, however, appear as an expression in the LENG() function.

2.7.2.9 FFDL Variables: HICD

This data unit outputs the current character's interlaced bitmap, in character priority mode (sections 2.7.2.3 and 2.7.2.18). The bitmap is output in two passes of odd/even rows (HICD comes from "Horizontal Interlaced Compact Data"). Only coloured dots are output. For each coloured dot, the exact coordinates are given.

This is a relatively complex data unit, designed to meet very unusual hardware and software needs. For example, the horizontal printing density of a Honeywell Bull 4/4x series printer, which requires such a data format, is about four times its vertical density. Fonts created for this printer are subjected to very restrictive rules regarding the number and displacement of coloured dots. When these specifications are met, the HICD data unit produces the most compact and efficient data outputs.

The first pass (in which odd-numbered rows are processed) is prefixed by a single 8-bit value indicating the length of the data (in bytes plus one) transmitted during that pass (e.g. a prefix of one would mean "zero bytes follow"). Each pass is divided into horizontal stripes, each addressing a maximum of 7 adjacent vertical printer pins. A byte prefixing each stripe indicates which pins are addressed within that stripe (bits 7-1), and whether other stripes follow (bit 0) in the same pass. Each stripe contains a sequence of horizontal coordinate values (leftmost position = 0) for each printer pin marked as "used" by that stripe's prefix. A sequence of coordinates referring to the same printer pin must be terminated by setting the 7th bit of the last coordinate.

The HICD data unit cannot be used as a term of an expression. The variable contains graphic data, which cannot be handled by the operators described in section 2.7.3. No prefix codes can be used before this variable. HICD may appear as an expression in the LENG() function.

2.7.2.10 FFDL Variables: KERN

This variable contains the offset from the left at which the graphic data of the current character is to appear. The variable is associated with the "Kerning" parameter described in section 3.6.

The meaning of the word "kerning" used in this documentation is inherited from the Amiga environment. Amiga fonts do not have a two-dimensional kerning table like those of other systems.

2.7.2.11 FFDL Variables: LAST

This variable is similar to FRST (section 2.7.2.7), only that it is relative to the last character on which the SQON and SQOF FFDL sequences

are repeated.

2.7.2.12 FFDL Variables: LCLM

This variable indicates the last column of the current character which contains at least one coloured dot. A value of XSIZ-1 may be used to indicate that the character is completely empty. A variable often used in combination with LCLM (last column) is FCLM (first column), which is described in section

2.7.2.6
.

2.7.2.13 FFDL Variables: LENG()

This data unit is complete only if the four-letter code is followed by an expression contained in parentheses. This is a function, rather than a variable. It calculates and outputs the length (in bytes) which would be occupied by the output of the FFDL sequence contained in the parentheses. No data associated with the FFDL sequence is actually output. If the FFDL sequence is also to be output, it must be repeated after the LENG(...) function. The first parenthesis must immediately follow the "LENG" keyword.

For example:

```
LENG(wXDPI)
```

equals 2 (one word = two bytes).

This function is useful whenever it is necessary to know in advance how many bytes will be occupied by the output of an FFDL sequence. LENG() is similar to the "sizeof()" operator of the C programming language.

2.7.2.14 FFDL Variables: REF1-4

This is a set of four variables: REF1, REF2, REF3 and REF4. They are associated with the "Reference Point" parameters described in sections

2.3
and 3.21.

2.7.2.15 FFDL Variables: REPT()

Like LENG(), this is a function. The name is an abbreviation of "REPeaT". Two expressions, separated by a comma sign (’,’), must be contained between two parentheses immediately following the four-letter code. No space may appear between "REPT" and the first ’(’.

The data unit causes the output of the data associated with the second expression for as many times as specified by the first expression. This means that the first expression specifies how many times the second expression is to be output. REPT() can be compared to the "for" statement which can be found in many computer programming languages. If the first

value is smaller than 1 (zero or negative), no data is output. Graphical data variables (e.g. CIDT, HDAT, HICD, VDAT and VIDT) cannot appear in the second expression of REPT().

To specify the output format of the second expression, the whole data unit must be prefixed with the desired prefix (as described in section

2.7.4

). For example, to output 10 "zero" words (20 bytes),

```
REPT((10),wNUL)
```

cannot be written, and would result in an error, while

```
wREPT((10),NUL)
```

would be correct, and

```
REPT((20),NUL)
```

would produce the same result.

One possible application of this function is the output of pad data to fill space. This may be necessary if a printer accepts only fixed-width characters, but a font which was designed as proportionally-spaced is to be output.

The following sequence is one of the FFDL sequences used to output draft mode characters in an 11 x 8 (width by height) matrix to an IBM Proprinter printer. It contains the REPT() function to add some "zero" bytes after the last column of dots. This makes sure that sequence always has a constant length. If the character is less than 11 columns wide, then some empty columns are added. The sequence is:

```
ESC \= x(1*13+2) (20) CNUM (0) (0) VDAT REPT((11-LENG(VDAT)),NUL)
```

This sequence is repeated for every active character in the selected range of the font, as described in section 7.3.6.

This function can also be used to output some data only if a particular condition is met. If the first expression is a logical expression, containing one or more relational operators (section

2.7.3.2

), the data associated with the second expression will only be output if the first expression is true.

For example:

```
sREPT((CNUM==(CNUM/10)*10),CNUM)
```

will output a string consisting of the digits which make up the ASCII code of CNUM, if (and only if) CNUM is a multiple of 10.

2.7.2.16 FFDL Variables: SPCE

This variable contains the "logical" horizontal space (opposed to the "physical" space, defined by the XSIZ variable) in dots occupied by the current character. The variable is associated with the "Space" parameter described in section 3.5.

SPCE indicates by how many dots the print position has to be moved to the right after the graphic data associated with the current character is printed.

2.7.2.17 FFDL Variables: VDAT

The VDAT data unit (from "Vertical DATa") causes the output of the current character's bitmap data as a sequence of columns, from the left to the right of the bitmap.

This data unit cannot be used as a term of an expression. The variable contains graphic data, which cannot be handled by the operators described in section
2.7.3
. No prefix codes can be used before this variable. VDAT can, however, appear as an expression in the LENG() function.

2.7.2.18 FFDL Variables: VIDT

This data unit is similar to CIDT (section
2.7.2.3
). It too, outputs interlaced graphical data in the form of vertical columns from left to right. While CIDT outputs the data in just two passes (character priority mode), VIDT (from "Vertical Interlaced DaTa") cycles through the odd/even steps at every column (column priority mode).

Like CIDT, HDAT, HICD and VDAT, VIDT cannot be used as a term of an expression. The variable contains graphic data, which cannot be handled by the operators described in section
2.7.3
. No prefix codes can be used before this variable. VIDT can, however, appear as an expression in the LENG() function.

2.7.2.19 FFDL Variables: XBYT

This variable indicates how many bytes are necessary to store one horizontal ("X") row of dots of the bitmap of the current character. If the number of bits (dots) is not a multiple of 8, the value is rounded up to the next multiple of 8 before it is converted into bytes (i.e. divided by 8).

2.7.2.20 FFDL Variables: XDPI

This variable is associated with the "Horizontal Dots Per Inch" parameter, described in section 7.3.3. The equivalent program parameter is described in section

2.6.23
.

Since the value may easily exceed 255, which is the maximum number which can be output by a standard data unit without prefix, the ’w’ prefix should be used to force the variable to be output as two bytes (see section
2.7.4
).

2.7.2.21 FFDL Variables: XMAX

This variable is associated with the parameter described in section 7.3.1. The equivalent program parameter is described in section
2.6.24
.

The maximum horizontal size of each character in the font, expressed in dots, is defined by the XMAX parameter. XMAX is not current character-relative.

2.7.2.22 FFDL Variables: XSIZ

This variable is associated with the "XSize" variable described in section 3.4. The variable contains the physical size occupied by the current character.

Unlike SPCE (section
2.7.2.16
), which determines the "logical" horizontal space occupied by the current character, XSIZ contains the exact width of the character matrix.

2.7.2.23 FFDL Variables: YBYT

This variable indicates the number of bytes which are necessary to store one vertical ("Y") column of dots of a character bitmap. This value is constant for all characters in the font, as all characters have the same height. If the number of bits (dots) per column is not a multiple of 8, the value is rounded up to the next multiple of 8 before it is converted into bytes (i.e. divided by 8).

2.7.2.24 FFDL Variables: YDPI

This variable is associated with the "Vertical Dots Per Inch" parameter, described in section 7.3.4. The equivalent program parameter is described in section
2.6.25
.

Since the value may easily exceed 255, which is the maximum number which can be output by a standard data unit without prefix, the ’w’ prefix should be used to force the variable to be output as two bytes (see

section
2.7.4
).

2.7.2.25 FFDL Variables: YMAX

This variable is associated with the parameter described in section 7.3.2. The equivalent program parameter is described in section
2.6.26
.

The maximum vertical size of each character in the font, expressed in dots, is defined by the YMAX parameter. YMAX is not current character-relative.

As all characters in the font have the same height, YMAX can be used as a synonym of YSIZ.

2.7.2.26 FFDL Variables: YSIZ

The Personal Fonts Maker does not use a "YSize" variable. This variable exists only to create a logical pair with XSIZ.

The height of the matrix of each character in a font is constant. Therefore, even if YSIZ is current character-relative, it is always equal to YMAX.

2.7.3 FFDL Operators

Whenever the Personal Fonts Maker encounters a left parenthesis in an FFDL sequence, it enters a formula parser mode. A formula (or expression) may contain constants (ASCII codes, characters, decimal and hexadecimal codes), variables (all variables except those which contain graphic data: CIDT, HDAT, HICD, VDAT and VIDT) and algebrical and logical operators.

Two parentheses and the whole expression between them can be used exactly like a variable, therefore also as a complete data unit. If the expression is part of another expression, it is not output as a data unit, but only used for calculations. Only the outermost level of parentheses is treated as a data unit.

The LENG() and REPT() functions (sections
2.7.2.13
and
2.7.2.15
)
perform an operation on the content of the parentheses following them. The parentheses following these funcions are linked to the functions, and are not considered independent data units.

Data unit prefixes (described in section
2.7.4
) cannot be used inside
the parentheses, but are accepted to define the output format of the whole

expression as a data unit. To do so, the prefix must come immediately before the initial left parenthesis.

The "0x" and '0' prefixes can (and must) always be used to prefix hexadecimal and octal numbers, respectively.

FFDL operators are very similar to the operators which are used in the C programming language, though C has some more operators. The following sections describe each operator in detail.

2.7.3.1 FFDL Algebrical Operators

There are five algebrical operators: '+' (plus), '-' (minus), '*' (multiply), '/' (divide) and '**' (power).

The result of the '+' operator is the sum of the operands, while the result of the '-' operator is the difference of the operands. The multiply and divide signs are used to perform standard multiplication and division operations.

The result of the '**' operator is the first operand to the power of the second operand, i.e. the first operand is multiplied by itself as many times as defined by the second operand minus one.

For example:

```
(4 ** 3)
```

equals 64 (four cubed).

The '-' (minus) sign can also be used as unary operator. A '-' sign after another operator or an open parenthesis indicates that the following value is negative (or positive, if '-' precedes a variable whose value is already negative). The unary plus sign ('+') is simply skipped by the Personal Fonts Maker, as it is not significant in the evaluation of an expression.

For example:

```
-5 + +4
```

equals -1. Examples of other valid expressions are:

```
(2 * -XMAX)
```

and

```
(-(CNUM + 1) * 10)
```

A detailed description of the priorities of the algebrical operators follows in section 2.7.3.4.

2.7.3.2 FFDL Relational Operators

These operators are called "relational" because they are used to determine the relationship between one operand (a variable or a constant) and another. These operators are used to compare two quantities, and modify some data accordingly.

The result of the '==' (equal to), '!=' (not equal to) '<' (less than), '>' (greater than), '<=' (less than or equal to) and '>=' (greater than or equal to), operators is always either 1 (one) or 0 (zero).

All relational operators yield 1 if the specified relation is true, and 0 if it is false.

The result of

```
(XSIZ > 8)
```

will be 1 if XSIZ is greater than 8, zero otherwise.

An operand cannot be shared by more than one operator. For example:

```
(\A <= CNUM <= \Z)
```

always yields 1 ("\A <= CNUM" can only yield 0 or 1; both values are smaller than \Z), while the result of

```
((\A <= CNUM) & (CNUM <= \Z))
```

which is the same as

```
((\A <= CNUM) == (CNUM <= \Z))
```

will be 1 if the ASCII code of CNUM corresponds to a capital letter between 'A' and 'Z', 0 otherwise.

2.7.3.3 FFDL Bit Manipulation Operators

The bit manipulation operators are used to operate on the single data bits which constitute the output stream. Section 1.3 introduces concepts like bit and byte.

There are five FFDL operators to manipulate bits. These are: '&' (bitwise AND), '^' (bitwise eXclusive OR = XOR), '|' (bitwise inclusive OR), '<<' (shift left) and '>>' (shift right).

The result of the AND operation is a value whose bits are set only at those positions in which the same bits of both operands are set. The OR operator produces a value which has all those bits set where the same bits are also set in either one or both operands. The bits in the result of the XOR operation are only set in those positions where the same bits in the two operands have different values (i.e. one is 0 and the other 1, or vice versa).

For example, the number 1 (one = binary 01) has only the first bit set. The number 2 (two = binary 10) has only the second bit set, while the number 3 (three = binary 11) has the first two bits set. Leading 0s

(zeroes) in binary numbers are used for readability only. Therefore:

```
(1 & 3)        yields 1,

(1 & 2)        yields 0,

(1 ^ 3)        yields 2,

(1 ^ 1)        yields 0,

(1 | 2)        yields 3 and

(1 | 3)        yields 3.
```

The '&', '^' and '|' operators are associative. More operands of the same operator may be rearranged. For example, the result of both

```
(\A | \B | 32)
```

and

```
(32 | \A | \B)
```

is the ASCII code of the lower case 'c' letter. A look at the ASCII character codes in appendix B may make this easier to understand.

These first three operators perform a logical AND, XOR and OR operation on each bit of their operands. Therefore, they can also be used as logical, rather than bitwise operators, if they are used to manipulate only data which can be either 0 or 1, like the result of the relational operators described in section
2.7.3.2
. In this case, the bitwise operators work only on one bit, becoming logical operators, so that the truth-value (0 or 1) of a multiple condition can be found. For example:

```
(REF1 < 8 & REF2 < 16)
```

can be read as "IF REF1 is smaller than 8 AND REF2 is smaller than 16 THEN the expression is TRUE (= 1), OTHERWISE it is FALSE (= 0)". More properly, it should be read as the bitwise AND operation of two values which can either be 0 or 1, therefore yielding 1 only if both values are 1 (i.e. REF1 < 8 and REF2 < 16) and 0 otherwise.

The shift operators move (shift) the bits which make up a value to the left or to the right, inside the byte (or word, as in section
2.7.4
) which contains them. The step (i.e. the number of bits) by which the data bits are shifted to the left or to the right is determined by the operand following the shift operator. The second operand cannot be negative, nor greater than or equal to the length, in bits, of the first operand (i.e. 8 or 16).

The '<<' (shift left) operator moves the bits which make up the content of the first operand to the left. In the process, bit 1 is changed to what was bit 0, bit 2 takes the previous content of bit 1, and so on. Bit 0 is

always cleared (set to 0). The last bit on the left is "lost". This is done as many times as specified by the second operand. Since all FFDL variables are internally represented with 16 bits, the "lost" bits can be recovered by a successive '>>' (shift right) operation, as long as they are not shifted out of the 16-bit range. The value referred to by the first operand is doubled as many times as specified by the second operand.

The '>>' (shift right) operator is similar to the other shift operator, only that it moves the bits to the right. The last bit on the left is cleared to 0 if the first operand is positive (or 0), and set to 1 otherwise (this preserves the sign of the first operand). Since the Personal Fonts Maker internally always uses 16 bits to store FFDL values, the leftmost bit is bit 15. The rightmost bit is lost. The first operand is halved the number of times specified by the the second operand.

For example:

```
(1 << 1)    yields 2  (01 shifted left by one bit becomes 10),
(1 << 2)    yields 4  (001 shifted left by two bits becomes 100),
(3 << 2)    yields 12 (0011 shifted left by two bits becomes 1100),
(1 >> 1)    yields 0  (1 shifted right by one bit becomes 0) and
(3 >> 1)    yields 1  (11 shifted right by one bit becomes 01).
```

The operators described in this section can be especially useful when several variables are to be "packed" into a single data unit. Some printers, for example, use four bits of a byte to store one value, and the remaining four for another value. This, of course, can only be done if the values do not exceed 15 (which is the maximum number which can be represented using four bits).

If, for example, the XBYT and YBYT variables are to be output as a single byte (XBYT being stored in the highmost four bits), the corresponding data unit must be described as:

```
b((XBYT << 4 ) | YBYT)
```

which could also be written as

```
(XBYT << 4 | YBYT)
```

since the "shift left" operator has a higher priority than the OR operator (section 2.7.3.4) and 'b' is the default data format (section 2.7.4). The first of the two examples is more explicit and readable, also because spaces were used to isolate variables.

2.7.3.4 FFDL Operator Priorities

Each FFDL operator has an associated priority, which is a value that can range from 1 to 11. The priority determines the order in which calculations are performed when there is more than one operator in an expression. Generally speaking, expressions are evaluated from the left to

the right (left-to-right associativity), but operators with higher priority are evaluated before those having lower priorities. Several operators with the same priority are also handled from the left to the right. The only exception is the unary minus operator (the unary plus is simply skipped), which has right-to-left associativity.

Parentheses (round brackets) force the formula parser to process the data between the parentheses as a unit, before the data outside the parentheses. Two parentheses and the data they contain can be used as a single operand. Parentheses can be nested. The inner levels of parentheses are processed before outer levels. The priority of a couple of parentheses is higher than that of any operator.

For example (the priority of '*' is higher than that of '+'):

```
(2 * 3 + 4)     yields 10,
(4 + 2 * 3)     yields 10, but
(2 * (3 + 4))   yields 14.
```

Parentheses in excess are allowed. This can be useful to make an FFDL sequence more readable. Once again it should be noted that only the outermost level of parentheses is handled as a data unit, and therefore output by the Personal Fonts Maker. The nesting of parentheses does not cause any "side effects".

```
((2 * ((3))) + (4))
```

gives the same result (10) as

```
(2 * 3 + 4)
```

To save parentheses, the operators are distributed among eleven levels of priorities:

Level 11: '(' and ')' parentheses (highest priority)

Level 10: '+' and '-' (unary plus and minus, having right-to-left associativity)

Level 9: '**' (power)

Level 8: '*' (multiply) and '/' (divide)

Level 7: '+' (plus) and '-' (minus)

Level 6: '<<' (shift left) and '>>' (shift right)

Level 5: '>' (greater than), '<' (less than), '>=' (greater than or equal to) and '<='(less than or equal to)

Level 4: '==' (equal to) and '!=' (not equal to)

Level 3: '&' (bitwise AND)

Level 2: '^' (bitwise XOR)

Level 1: '|' (bitwise OR).

The expression

```
((2+4) > (1+3))
```

can be written as

```
( 4 + 2 > 3 + 1 )
```

yielding the same result (1).

2.7.4 FFDL Prefixes

Data units can be prefixed by a single character to force the data to be output in a certain format. This operation is generally referred to as casting. By default, each data data unit is output as a single byte (8-bits, signed).

The prefix must immediately precede the data unit, be it a constant, a variable or an expression delimited by parentheses.

The following prefixes are accepted:

b: The data unit is output as a signed 8-bit byte. This is the default format.

w: The data unit is output as a signed word (16 bits), consisting of the high byte followed by the low byte (the format used by Motorola microprocessors).

x: The data unit is output as a signed word in the format used by Intel microprocessors, with the low byte before the high byte.

s: The data unit is output as a signed decimal string (i.e. a series of digits from '0' to '9', ASCII codes 48 to 57). The number must be in the range from -32768 to 32767. A '-' sign is prefixed to negative numbers.

u: The data unit is output as an unsigned decimal string. The number must be in the range from 0 to 65535.

$: The data unit is output as a hexadecimal string (i.e. a series of digits from '0' to 'F', ASCII codes 48 to 57 and 65 to 70). The number must be in the range from 0 to FFFF (65535 decimal). This is similar to the "0x" prefix used to define FFDL constants, only that it is used for data output.

The 'b', 'w' and 'x' prefixes cause all values to be output in the "signed" format. This means that one bit is used to indicate whether the remaining bits represent a positive or a negative number.

Most computer microprocessors have adopted the two's complement arithmetic to represent negative numbers. The most significant bit determines the sign (0 = positive, 1 = negative). In simple words, the rule to change the sign is "flip the bits and add 1". This does not cover some special cases, but is sufficient for the following example:

```
b(1)          is output as binary 00000001, and
b(-1)         is output as binary 11111111, but
b(255)        is also output as 11111111.
```

The 'b' prefix was not necessary in the above examples, as it is the default format. The example should make clear how the interpretation of the values is up to the program or device receiving the data. It is not possible to distinguish between a group of n bits from a group of n-1 bits plus a sign bit. Therefore, if a negative number is output to a device which expects only positive numbers in its input stream, the value will be interpreted as positive.

No negative values should be output if the recipient cannot handle them. This latter case implies that it is not necessary to output negative values. Conversely, if the recipient handles negative numbers, the most significant bit must only be set if the value is negative. Otherwise, a -1 can be interpreted as 255, or vice versa.

The 's', 'u' and '$' prefixes cause the data unit to be output as a sequence of ASCII digits. No zero (NUL) code is output after the last character. If this is requested by the recipient to distinguish the end of the string, it must be hand coded in FFDL by adding a (0).

Examples:

```
(1+2*5)            yields 11 (byte).
s(100/(-2+4))      yields 50 (decimal string: "50").
w(CR+LF)           yields 23 (word: CR = 13 + LF = 10).
$((5+4)-(15-5))    yields FFFF (hexadecimal string: "FFFF").
b(2 * XSIZ)        yields 60 (byte) if XSIZ is 30.
(0x10-0x6)         yields 10 (byte).
x(LENG($(255))*2)  yields 4 (exception word: byte 4 + byte 0).
```

2.8 Character Sets

Character sets are another aspect of fonts. Character sets have nothing to do with the graphical and artistical part of the font. Instead, a character set determines the order in which the signs appear in the font. A character set defines which letters, figures, symbols, and other signs are to be part of a font, and the position of the signs in the font.

Each character in a font has a unique code, which indicates the position of the character in the font. The code indicates the position of each character as an offset from the first character. The code of the first character is 0 (zero). The code of the second character is 1 (one), and so on.

Unfortunately, there is not one universal character set. There are different character sets for different countries, computers, programs and printers. If there were one standard character set, all fonts would

provide the same collection of characters. But different character sets does not necessarily mean different fonts. The characters in a font can be re-arranged to match the codes of different character sets. The Personal Fonts Maker can do this automatically.

The minimal character set for most fonts is the 7-bit ASCII set, originally developed for the North-American market. This character set has 96 characters, which include all letters, numbers and other symbols which can be found on a standard (US) computer terminal or typewriter. The ASCII character set also has 32 special codes reserved for control characters, like LF (Line Feed) and FF (Form Feed). The sum of 96 and 32 is 128, which is the maximum number of different codes which can be represented with 7 bits.

It is easy to imagine that neither 96 nor 128 characters can be sufficient for all possible user requirements. Variants of the US 7-bit ASCII set were developed to match different user needs. Little-used characters of the US set, like the backslash ('\'), some braces and brackets ('{', '}', '[' and ']') and other signs were replaced with the characters needed by different groups of users. In the Legal set, for example, the "registered", "trademark" and "copyright" symbols replaced other signs. Different national character sets replace unused characters with local signs, like accented letters.

Since it is often not practicable to select a new character set to write a sign which is not part of the few of a particular variant of the US ASCII set, extended (or composite) character sets with up to 192 or 256 characters were provided. The Personal Fonts Maker can deal with fonts and character sets with as many as 256 codes (plus one, as explained in the following paragraphs). As for 7-bit character sets, no standard for 8-bit extended character sets exists.

The Personal Fonts Maker comes with different character set files which cover most 7-bit character sets and several extended sets.

When standard Amiga characters are displayed on the Personal Fonts Maker screen, on the title bar or to show FFDL sequences or parameters in string gadgets, the default Amiga character set is used. Most printers, on the other hand, adopt the IBM PC 8-bit character set, and/or the Epson 7-bit national character sets. Appendixes C, D and E show the differences between these character sets.

Most fonts which come with the Personal Fonts Maker use the original IBM PC character set. Amiga fonts, and the fonts which can be saved and read by the Personal Fonts Maker in this format, use the Amiga character set.

The Personal Fonts Maker can be programmed to use different character sets (see sections 4.7 to 4.11). A character set can be designed, saved and loaded again at any time. Each font environment (sections
2.6
,
2.6.8
and 3.2) can use a different character set.

A character set file contains graphic data like a font. This data is used for the default character representation of each character (section

3.7). This is necessary to show the user what character is currently being edited, using a complete default picture of the character. The Amiga fonts cannot be used to do this, as some character sets have signs which are not contained in the Amiga character set.

A character set file also contains an encoding vector. This is a programmable conversion table, used to exchange non-Amiga fonts with the Amiga font handling routines. This flexibility in character encoding is valuable for two reasons. First, it allows the user to work with fonts in which the characters are not ordered following the standard Amiga set, like printer fonts adopting the PC set. Second, it makes it possible to import and export font data between the Amiga fonts and fonts with a different encoding.

As described in section 4.10, the data in the encoding vector can be programmed by the user. For each character in a set to be described, the equivalent Amiga code must be written. The positions in the encoding vector for which the Amiga has no equivalent character must be filled with a "-1". These characters are not translated during the exchange of Amiga font data with fonts arranged according to a different character set.

The character set of the Amiga and the other sets which can be used by the Personal Fonts Maker define a maximum of 256 characters. Amiga font files, however, always contain graphical data for an additional character. This is the undefined character. The undefined character is displayed by all programs, including the Amiga operating system, whenever no graphic data has been defined for a code to be displayed. The Amiga default undefined character is a rectangle similar to a squared ’O’ letter. The Personal Fonts Maker also allows the editing of this additional character, which has the code 256 (it is, thus, the 257th character counting from 1). The undefined character is always used when data is exchanged in the Amiga font format. It can also be saved with fonts in the PFM format. The undefined character cannot, however, be accessed by the FFDL, nor can it appear in the character set encoding vector.

The standard ASCII set table appears in appendix B. Appendix C contains the default codes adopted by the Personal Fonts Maker and most printers. Appendix D lists the Amiga character set, while appendix E contains different 7-bit character sets. Section 13.3 describes how to create an extended font combining the characters of an existing font.