

The FIGfont Version 2 FIGfont and FIGdriver Standard

Draft 2.0 Copyright 1996, 1997 by John Cowan and Paul Burton
 Portions Copyright 1991, 1993, 1994 by Glenn Chappell and Ian Chai
 May be freely copied and distributed.



- [Introduction](#)
- [Basic Definitions and Concepts](#)
 - ["FIGfont"](#)
 - ["FIGcharacters" and "Sub-characters"](#)
 - ["FIGdriver"](#)
 - ["FIGure"](#)
 - ["FIG"](#)
 - ["Layout Modes"](#)
 - ["Smushing Rules"](#)
 - ["Hardblanks"](#)
 - ["Character Sets" and "Character Codes"](#)
- [Creating FIGfonts](#)
 - [The Header Line](#)
 - [Interpretation of Layout Parameters](#)
 - [Setting Layout Parameters Step-by-Step](#)
 - [FIGfont Comments](#)
 - [FIGcharacter Data](#)
 - [Basic Data Structure](#)
 - [Required FIGcharacters](#)
 - [Code Tagged FIGcharacters](#)
- [Notes - Avoiding Errors and General Advice](#)
- [Control Files](#)
 - [Standard Format](#)
 - [Extended Commands](#)
- [Standardized Capabilities of current and future FIGdrivers](#)
- [Chart of Capabilities of FIGlet 2.2 and FIGWin 1.0](#)

Introduction

This document specifies the format of font files, and the associated control files, used by the FIGlet and FIGWin programs (FIGdrivers). It is written for designers who wish to build fonts (FIGfonts) usable by either program, and also serves as a standard for development of future versions or similar FIGdrivers. Some features explained here are not supported by both programs. See separate documentation to learn how to use FIGlet or FIGWin.

NOTE: FIGWin 1.0 is packaged with a program called FIGfont Editor for Windows 1.0, which is just that. It does not require a complete understanding of this document to create FIGfonts. However it is a good idea to become familiar with the "BASIC DEFINITIONS AND CONCEPTS" information before using it.

If you design a FIGfont, please send an e-mail announcement to <figletfonts@onelist.com>, the FIGlet fonts mailing list, and email a copy to <ianchai@usa.net> for him to put it at the ftp site.

Basic Definitions and Concepts

"FIGfont"

A FIGfont is a file which represents the graphical arrangement of characters representing larger characters. Since a FIGfont file is a text file, it can be created with any text editing program on any platform. The filename of a FIGfont file must end with ".flf", which stands for "FIGLettering Font".

"FIGcharacters" and "Sub-characters"

Because FIGfonts describe large characters which consist of smaller characters, confusion can result when discussing one or the other. Therefore, the terms "FIGcharacter" and "sub-character" are used, respectively.

"FIGdriver"

The term FIGdriver is used in this document to encompass FIGlet, FIGWin, and any future programs which use FIGfonts.

"FIGure"

A FIGure (thusly capitalized) is an image created by a FIGdriver.

"FIG"

A bit of history:

In Spring 1991, inspired by the Email signature of a friend named Frank, and goaded on by Ian Chai, Glenn Chappell wrote a nifty little 170-line "C" program called "newban", which would create large letters out of ordinary text characters. At the time, it was only compiled for UNIX. In hindsight, we now call it "FIGlet 1.0". FIGlet stands for **F**rank, **I**an, and **G**lenn's **l**etters. In various incarnations, newban circulated around the net for a couple of years. It had one font, which included only lowercase letters.

In early 1993, Ian decided newban was due for a few changes, so together Ian and Glenn added the full ASCII character set, to start with. First, though, Ian had to find a copy of the source, since Glenn had tossed it away as not worth the disk space. Ian and Glenn discussed what could be done with it, decided on a general re-write, and, 7 months later, ended up with 888 lines of code, 13 FIGfonts and documentation. This was FIGlet 2.0, the first real release.

To their great surprise, FIGlet took the net by storm. They received floods of "FIGlet is great!" messages and a new contributed FIGfont about once a week. To handle all the traffic, Ian quickly set up a mailing list, Daniel Simmons kindly offered space for an FTP site, several people volunteered to port FIGlet to non-Unix operating systems, ...and bug reports poured in.

Because of these, and the need to make FIGlet more "international", Ian and Glenn released a new version of FIGlet which could handle non-ASCII character sets and right-to-left printing. This was FIGlet 2.1, which, in a couple of weeks, became figlet 2.1.1. This weighed in at 1314 lines, and there were over 60 FIGfonts.

By late 1996, FIGlet had quite a following of fans subscribing to its mailing list. It had been ported to MS-DOS, Macintosh, Amiga, Apple II GS, Atari ST, Acorn and OS/2. FIGlet had been further updated, and there were nearly 200 FIGfonts.

John Cowan and Paul Burton are two FIGlet fans who decided to create new versions. While John wrote FIGlet version 2.2 using C, Paul wrote FIGWin 1.0, the first true GUI (Windows) implementation of FIGlet, using Visual Basic. John and Paul worked together to add new features to FIGfont files which could be read by both programs, and together wrote this document, which we hope helps to establish consistency in FIGfonts and help with the creation of future FIGdrivers. FIGlet 2.2 has about 4800 lines of code, of which over half is a support library for reading compressed files.

FIGlet 2.2 and FIGWin 1.0 both allow greater flexibility by use of new information which can be contained in FIGfont files without interfering with the function of older FIGdrivers.

NOTE: The Macintosh version of FIGlet is still command-line driven as of this writing, and a GUI version is very much in demand. The FIGlet C code is written to be easily plugged in to a GUI shell, so it will be a relatively easy task for a Macintosh developer.

"Layout Modes"

A FIGdriver may arrange FIGcharacters using one of three "layout modes", which define the spacing between FIGcharacters. The layout mode for the horizontal axis may differ from the layout mode for the vertical axis. A default choice is defined for each axis by every FIGfont.

The three layout modes are:

- Full Size (Separately called "Full Width" or "Full Height".)
Represents each FIGcharacter occupying the full width or height of its arrangement of sub-characters as designed.
- Fitting Only (Separately called "Kerning" or "Vertical Fitting".)
Moves FIGcharacters closer together until they touch. Typographers use the term "kerning" for this phenomenon when applied to the horizontal axis, but fitting also includes this as a vertical behavior, for which there is apparently no established typographical term.

- Smushing (Same term for both axes.)

Moves FIGcharacters one step closer after they touch, so that they partially occupy the same space. A FIGdriver must decide what sub-character to display at each junction. There are two ways of making these decisions: by controlled smushing or by universal smushing.

Controlled smushing uses a set of "smushing rules" selected by the designer of a FIGfont. (See ["Smushing Rules"](#) below.) Each rule is a comparison of the two sub-characters which must be joined to yield what to display at the junction. Controlled smushing will not always allow smushing to occur, because the compared sub-characters may not correspond to any active rule. Wherever smushing cannot occur, fitting occurs instead.

Universal smushing simply overrides the sub-character from the earlier FIGcharacter with the sub-character from the later FIGcharacter. This produces an "overlapping" effect with some FIGfonts, wherein the latter FIGcharacter may appear to be "in front".

A FIGfont which does not specify any smushing rules for a particular axis indicates that universal smushing is to occur when smushing is requested. Therefore, it is not possible for a FIGfont designer to "forbid" smushing. However there are ways to ensure that smushing does not cause a FIGfont to be illegible when smushed. This is especially important for smaller FIGfonts. (See ["Hardblanks"](#) for details.)

For vertical fitting or smushing, entire lines of output FIGcharacters are "moved" as a unit.

Not all FIGdrivers do vertical fitting or smushing. At present, FIGWin 1.0 does, but FIGlet 2.2 does not. Further, while FIGlet 2.2 allows the user to override the FIGfont designer's set of smushing rules, FIGWin 1.0 does not.

NOTE: In the documentation of FIGlet versions prior to 2.2, the term "smushmode" was used to mean the layout mode, and this term further included the smushing rules (if any) to be applied. However, since the layout mode may or may not involve smushing, we are straying from the use of this somewhat misleading term.

"Smushing Rules"

Again, smushing rules are for controlled smushing. If none are defined to be active in a FIGfont, universal smushing occurs instead.

Generally, if a FIGfont is "drawn at the borders" using sub-characters "-_|\^[\]{}()<>", you will want to use controlled smushing by selecting from the rules below. Otherwise, if your FIGfont uses a lot of other sub-characters, do not select any rules and universal smushing will occur instead. (See ["Hardblanks"](#) below if your FIGfont is very small and would become illegible if smushed.) Experimentation is the best way to make these decisions.

There are six possible horizontal smushing rules and five possible vertical smushing rules. Below is a description of all of the rules.

NOTE: Ignore the "code values" for now. They are explained later.

The Six Horizontal Smushing Rules

Rule	Name	code value	Description
1	EQUAL CHARACTER SMUSHING	1	Two sub-characters are smushed into a single sub-character if they are the same. This rule does not smush hardblanks. (See "Hardblanks" below.)
2	UNDERSCORE SMUSHING	2	An underscore ("_") will be replaced by any of: " ", "/", "\", "[", "]", "{", "}", "(", ")", "<", or ">".
3	HIERARCHY SMUSHING	4	A hierarchy of six classes is used: " ", "\", "[", "]", "{", "()", and "<>". When two smushing sub-characters are from different classes, the one from the latter class will be used.
4	OPPOSITE PAIR SMUSHING	8	Smushes opposing brackets ("[" or "]", "{", "}" and "(", ")") together, replacing any such pair with a vertical bar (" ").
5	BIG X SMUSHING	16	Smushes "\^" into "Y", "/" into "X", and "<>" into "X". Note that "<>" is not smushed in any way by this rule. The name "BIG X" is historical; originally all three pairs were smushed into "X".
6	HARDBLANK SMUSHING	32	Smushes two hardblanks together, replacing them with a single hardblank. (See "Hardblanks" below.)

The Five Vertical Smushing Rules

Rule	Name	code value	Description
1	EQUAL CHARACTER SMUSHING	256	Same as horizontal smushing rule 1.
2	UNDERSCORE SMUSHING	512	Same as horizontal smushing rule 2.
3	HIERARCHY SMUSHING	1024	Same as horizontal smushing rule 3.
4	HORIZONTAL LINE SMUSHING	2048	Smushes stacked pairs of "-" and "_", replacing them with a single "=" sub-character. It does not matter which is found above the other. Note that vertical smushing rule 1 will smush IDENTICAL pairs of horizontal lines, while this rule smushes horizontal lines consisting of DIFFERENT sub-characters.
5	VERTICAL LINE SUPERSMUSHING	4096	This one rule is different from all others, in that it "supersmushes" vertical lines consisting of several vertical bars (" "). This creates the illusion that FIGcharacters have slid vertically against each other. Supersmushing continues until any sub-characters other than " " would have to be smushed. Supersmushing can produce impressive results, but it is seldom possible, since other sub-characters would usually have to be considered for smushing as soon as any such stacked vertical lines are encountered.

"Hardblanks"

A hardblank is a special sub-character which is displayed as a blank (space) in rendered FIGures, but is treated more like a "visible" sub-character when fitting or smushing horizontally. Therefore, hardblanks keep adjacent FIGcharacters a certain distance apart.

NOTE: Hardblanks act the same as blanks for vertical operations.

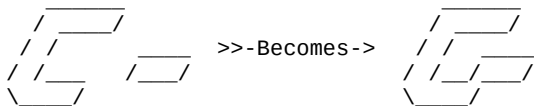
Hardblanks have three purposes:

1. Hardblanks are used to create the blank (space) FIGcharacter.

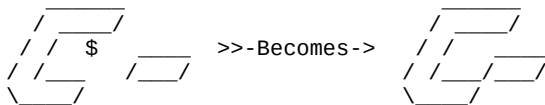
Usually the space FIGcharacter is simply one or two vertical columns of hardblanks. Some slanted FIGfonts as shown below have a diagonal arrangement of hardblanks instead.

2. Hardblanks can prevent "unreasonable" fitting or smushing.

Normally when fitting or smushing, the blank (space) sub-character is considered "vacant space". In the following example, a capital "C" FIGcharacter is smushed with a "minus" FIGcharacter.



The FIGure above looks like a capital G. To prevent this, a FIGfont designer might place a hardblank in the center of the capital C. In the following example, the hardblank is represented as a "\$":



Using hardblanks in this manner ensures that FIGcharacters with a lot of empty space will not be unreasonably "invaded" by adjacent FIGcharacters. Generally, FIGcharacters such as capital C, L or T, or small punctuation marks such as commas, may contain hardblanks, since they may contain a lot of vacant space which is "accessible" from either side.

3. Hardblanks can prevent smushing from making FIGfonts illegible.

This legitimate purpose of hardblanks is often overused. If a FIGfont designer is absolutely sure that smushing "visible" sub-characters would make their FIGfont illegible, hardblanks may be positioned at the end of each row of sub-characters, against the visible sub-characters, creating a barrier.

With older FIGdrivers, using hardblanks for this purpose meant that FIGcharacters would have to be separated by at least one blank in output FIGures, since only a hardblank could smush with another hardblank. However with the advent of universal smushing, this is no longer necessary. Hardblanks ARE overridden by any visible sub-character when performing universal smushing. Hardblanks still represent a "stopping point", but only AFTER their locations are occupied.

NOTE: Earlier it was stated that universal smushing overrides the sub-character from the former FIGcharacter with the sub-character from the latter FIGcharacter. Hardblanks (and blanks or spaces)

are the exception to this rule; they will always be overridden by visible sub-characters, regardless of which FIGcharacter contains the hardblank. This ensures that no visible sub-characters "disappear".

Therefore, one can design a FIGfont with a default behavior of universal smushing, while the output FIGure would LOOK like the effect of fitting, or even full size if additional hardblanks are used. If a user "scales down" the layout mode to fitting, the result would look like "extra spacing" between FIGcharacters.

Taking this concept further, a FIGcharacter may also include extra blanks (spaces) on the left side of each FIGcharacter, which would define the FIGcharacter's width as slightly larger than required for the visible sub-characters and hardblanks. With such a FIGfont, a user who further "scales down" the layout mode to full size would see even greater spacing.

These techniques prevent horizontal smushing from causing a FIGfont to become illegible, while offering greater flexibility of output to users.

NOTE: These techniques cannot be used to prevent vertical smushing of visible sub-characters, since hardblanks are not respected in the vertical axis. Although it is possible to select only one vertical smushing rule which involves only sub-characters which are not used in your FIGfont, it is recommend that you do NOT do so. In our opinion, most users would prefer to get what they ask for, rather than being told, in effect: "I, the FIGfont designer, have decided that you wouldn't like the results of vertical smushing, so I have prevented you from trying it." Instead, we recommend setting the default behavior to either fitting or full height, and either allowing universal smushing, or selecting vertical smushing rules which seem most appropriate. A user of your FIGfont will quickly see why you did not choose smushing as the default vertical layout mode, and will agree with you.

"Character Sets" and "Character Codes"

When you type using your keyboard, you are actually sending your computer a series of numbers. Each number must be interpreted by your computer so that it knows what character to display. The computer uses a list of definitions, called a "character set". The numbers which represent each character are called "character codes".

There are many character sets, most of which are internationally accepted as standards. By far, the most common character set is ASCII, which stands for "American Standard Code for Information Interchange". ASCII identifies its characters with codes ranging from 0 to 127.

NOTE: The term "ASCII art" has become well-understood to mean artistic images which consist of characters on your screen (such as FIGures).

For a list of the printable ASCII characters with the corresponding codes, see the section [Required FIGcharacters](#) below. The other ASCII codes in the range of 0 through 31 are "control characters" such as carriage-return (code 13), linefeed/newline (code 10), tab (code 9), backspace (code 8) or null (code 0). Code 127 is a delete in ASCII.

Getting more technical for just a moment: A byte consisting of 8 bits (eight 1's or 0's) may represent a number from 0 to 255. Therefore, most computers have DIRECT access to 256 characters at any given time. A character set which includes 256 characters is called an 8-bit character set.

For Latin-based languages, ASCII is almost always the first half of a larger 8-bit character set. Latin-1 is the most common example of an 8-bit character set. Latin-1 includes all of ASCII, and adds characters with codes from 128 to 255 which include umlauted ("double-dotted") letters and characters with various other accents. In the United States, Windows and most Unix systems have Latin-1 directly available.

Most modern systems allow the possibility of changing 8-bit character sets. On Windows systems, character sets are referred to as "code pages". There are many other character sets which are not mentioned here. DOS has its own character set (which also has international variants) that includes graphics characters for drawing lines. It is also an extension of ASCII.

For some languages, 8-bit character sets are insufficient, particularly on East Asian systems. Therefore, some systems allow 2 bytes for each character, which multiplies the 256 possibilities by 256, resulting in 65536 possible characters. (Much more than the world will ever need.)

Unicode is a character set standard which is intended to fulfill the worldwide need for a single character set which includes all characters used worldwide. Unicode includes character codes from 0 to 65535, although at present, only about 22,000 characters have been officially assigned and named by the Unicode Consortium. The alphabets and other writing systems representable with Unicode include all Latin-alphabet systems, Greek, Russian and other Cyrillic-alphabet systems, Hebrew, Arabic, the various languages of India, Chinese, Japanese, Korean, and others. The existing Unicode symbols include chess pieces, astrological signs, gaming symbols, telephones, pointing fingers, etc. --- just about any type of FIGcharacter you may wish to create. Unicode is constantly (but slowly) being extended to handle new writing systems and symbols. Information on Unicode is available at <http://www.unicode.org> and at <ftp://unicode.org>.

Unicode, Latin-1, and ASCII all specify the same meanings for overlapping character codes: ASCII 65 = Latin-1 65 = Unicode 65 = "A", formally known as "LATIN CAPITAL LETTER A".

Since a keyboard usually has only about 100 keys, your computer may contain a program called a "keyboard map", which will interpret certain keystrokes or combinations of keystrokes as different character codes. Keyboard maps use "mapping tables" to make these determinations. The appropriate keyboard activity for a given character code may involve several keystrokes. Almost all systems are capable of handling at least 8-bit character sets (containing 256 characters), so there is always an active keyboard map, at least for those characters which are not actually painted on the keys. (United States users may not even know that their computer can interpret special keystrokes. Such keystrokes may be something similar to holding down the ALT key while typing a character code on the numeric keypad. Try it!)

Below are characters 160 through 255, AS REPRESENTED ON YOUR SYSTEM.

¡ ¢ £ ¤ ¥ ¦ § ¨ © ª « ¬ ® ¯ ° ± ² ³ ´ µ ¶ · ¸ ¹ º » ¼ ½ ¾ ¿ À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï
Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß à á â ã ä å æ ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ÷ ø ù ú û ý þ ÿ

IMPORTANT NOTE: Depending on which character set is active on your system, you may see different characters. This document (like all computer documents) does not contains characters per se, only bytes. What you see above is your particular computer's representation of these byte values. In other words, your active character set. However, if it is Latin-1, the first visible character is an inverted "!", and the last is an umlauted "y". Although we can safely assume your computer has ASCII, it does not necessarily have the Latin-1 character set active.

What does all this have to do with FIGfonts???

First, it should be evident that it is best to use only ASCII characters for sub-characters when possible. This will ensure portability to different platforms.

FIGlet has gained international popularity, but early versions were made to handle only FIGcharacters with assigned character codes corresponding to ASCII. So, over the years there have been four methods used to create "virtual mapping tables" within the program itself:

The first method was simply to create FIGcharacters which do not look like the ASCII character set implies. For example, a FIGfont might contain Greek letters, and within its comments, it may say, "If you type A, you'll get a Greek Alpha" etc. With the advent of newer features, it is preferable not to use this method. Instead, when possible, add new FIGcharacters to existing FIGfonts or create new FIGfonts with FIGcharacters coded to match the expectations of ASCII/Latin-1/Unicode, and create an appropriate control file. (See [Control Files](#) below.) Remember that Unicode includes almost any character for which you may want to create a FIGcharacter.

The second method was very specific, to accommodate the German audience. A special option was added to the FIGlet program which would re-route input characters "[", "\", and "]" to umlauted A, O and U, while "{", "|", and "}" would become the respective lowercase versions of these. Also, "~" was made to become the s-z character when this special option was used. This was called "the -D option." The addition of this feature meant that all compatible FIGfonts must contain these Deutsch (German) FIGcharacters, in addition to the ASCII FIGcharacters. Although this option is still available in the most recent version, it is no longer necessary, as the same result can be achieved by the newer features described below. However, the requirement for Deutsch FIGcharacters remains for backward compatibility. (Or at least zero-width FIGcharacters in their place.)

Later, FIGlet was made to accept control files, which are quite literally a form of mapping table. (See [Control Files](#) below.) This was a significant advance for internationalization.

FIGlet 2.2 can now accept specially encoded formats of input text which imply more than one byte per character.

Creating FIGfonts

NOTE: FIGWin 1.0 is packaged with a program called FIGfont Editor for Windows 1.0, which is just that. There is no need to read further if you intend to use it. However, the section "CONTROL FILES" below is still relevant.

Since a FIGfont file is a text file, it can be created with any text editing program on any platform, and will still be compatible with FIGdrivers on all operating systems, except that the bytes used to indicate the end of each text line may vary. (PC's use carriage return and linefeed at the end of each line, Macintosh uses carriage return only, and UNIX uses linefeed only.)

This minor difference among operating systems is handled easily by setting your FTP program to ASCII mode during upload or download. So there is no need to be concerned about this as long as you remember to do this during file transfer.

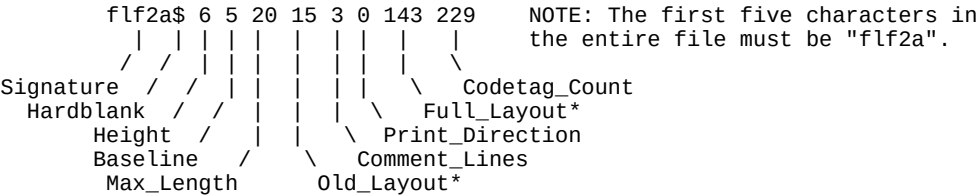
The filename of a FIGfont file must end with ".flf", which stands for "FIGLettering Font". The first part of the filename should contain only letters, and should be lowercase on operating systems which permit case sensitive filenames. The filename should be unique in the first 8 characters, since some older file systems truncate longer filenames.

It is easier to modify an existing FIGfont than it is to create a new one from scratch. The first step is to read and understand this document. You may want to load "standard.flf" or another FIGfont into a text editor as an example while you read.

A FIGfont file contains three portions: a header line, comments, and FIGcharacter data.

The Header Line

The header line gives information about the FIGfont. Here is an example showing the names of all parameters:



* The two layout parameters are closely related and fairly complex.
(See [Interpretation of Layout Parameters](#).)

For those desiring a quick explanation, the above line indicates that this FIGfont uses "\$" to represent the hardblank in FIGcharacter data, it has FIGcharacters which are 6 lines tall, 5 of which are above the baseline, no line in the FIGfont data is more than 20 columns wide, the default horizontal layout is represented by the number 15, there are 3 comment lines, the default print direction for this FIGfont is left-to-right, a complete description of default and possible horizontal and vertical layouts is represented by the number 143, and there are 229 code-tagged characters.

The first seven parameters are required. The last three (Direction, Full_Layout, and Codetag_Count, are not. This allows for backward compatibility with older FIGfonts, but a FIGfont without these parameters would force a FIGdriver to "guess" (by means not described in this document) the information it would expect to find in Full_Layout. For this reason, inclusion of all parameters is strongly recommended.

Future versions of this standard may add more parameters after Codetag_Count.

A description of each parameter follows:

Signature	The signature is the first five characters: "flf2a". The first four characters "flf2" identify the file as compatible with FIGlet version 2.0 or later (and FIGWin 1.0). The "a" is currently ignored, but cannot be omitted. Different characters in the "a" location may mean something in future versions of this standard. If so, you can be sure your FIGfonts will still work if this character is "a".
Hardblank	<div>Immediately following the signature is the hardblank character. The hardblank character in the header line defines which sub-character will be used to represent hardblanks in the FIGcharacter data.</div> <div>By convention, the usual hardblank is a "\$", but it can be any character except a blank (space), a carriage-return, a newline (linefeed) or a null character. If you want the entire printable ASCII set available to use, make the hardblank a "delete" character (character code 127). With the exception of delete, it is inadvisable to use non-printable characters as a hardblank.</div>
Height	The Height parameter specifies the consistent height of every FIGcharacter, measured in sub-characters. Note that ALL FIGcharacters in a given FIGfont have the same height, since this includes any empty space above or below. This is a measurement from the top of the tallest FIGcharacter to the bottom of the lowest hanging FIGcharacter, such as a lowercase g.
Baseline	<div>The Baseline parameter is the number of lines of sub-characters from the baseline of a FIGcharacter to the top of the tallest FIGcharacter. The baseline of a FIGfont is an imaginary line on top of which capital letters would rest, while the tails of lowercase g, j, p, q, and y may hang below. In other words, Baseline is the height of a FIGcharacter, ignoring any descenders.</div> <div>This parameter does not affect the output of FIGlet 2.2 or FIGWin 1.0, but future versions or other future FIGdrivers may use it. The Baseline parameter should be correctly set to reflect the true baseline as described above. It is an error for Baseline to be less than 1 or greater than the Height parameter.</div>

Max_Length	The Max_Length parameter is the maximum length of any line describing a FIGcharacter. This is usually the width of the widest FIGcharacter, plus 2 (to accommodate endmarks as described later.) However, you can (and probably should) set Max_Length slightly larger than this as a safety measure in case your FIGfont is edited to include wider FIGcharacters. FIGlet (but not FIGWin 1.0) uses this number to minimize the memory taken up by a FIGfont, which is important in the case of FIGfonts with many FIGcharacters.
Old_Layout	See Interpretation of Layout Parameters
Comment_Lines	Between the first line and the actual FIGcharacters of the FIGfont are the comment lines. The Comment_Lines parameter specifies how many lines there are. Comments are optional, but recommended to properly document the origin of a FIGfont.
Print_Direction	The Print_Direction parameter tells which direction the font is to be printed by default. A value of 0 means left-to-right, and 1 means right-to-left. If this parameter is absent, 0 (left-to-right) is assumed. Print_Direction may not specify vertical print, although FIGdrivers are capable of vertical print. Versions of FIGlet prior to 2.1 ignore this parameter.
Full_Layout	See Interpretation of Layout Parameters just below.
Codetag_Count	Indicates the number of code-tagged (non-required) FIGcharacters in this FIGfont. This is always equal to the total number of FIGcharacters in the font minus 102. This parameter is typically ignored by FIGdrivers, but can be used to verify that no characters are missing from the end of the FIGfont. The chkfont program will display the number of codetagged characters in the FIGfont on which it is run, making it easy to insert this parameter after a FIGfont is written.

Interpretation of Layout Parameters

Full_Layout describes ALL information about horizontal and vertical layout: the default layout modes and potential smushing rules, even when smushing is not a default layout mode.

Old_Layout does not include all of the information desired by the most recent FIGdrivers, which is the inspiration for the creation of the new Full_Layout parameter. Old_Layout is still required for backward compatibility, and FIGdrivers must be able to interpret FIGfonts which do not have the Full_Layout parameter. (See [Standardized Capabilities of current and future FIGdrivers.](#))

Versions of FIGlet prior to 2.2 do not recognize the Full_Layout parameter. Documentation accompanying FIGlet versions prior to 2.2 refer to Old_Layout as "smushmode", which is somewhat misleading since it can indicate layout modes other than smushing.

Old_Layout and Full_Layout must contain some redundant information.

Setting the layout parameters is a matter of adding numbers together ("code values"). What follows is a chart of the meanings of all code values. (You may skip down to [Setting Layout Parameters Step-by-Step](#) if you prefer, or if you find this portion confusing.)

Full_Layout: (Legal values 0 to 32767)

1	Apply horizontal smushing rule 1 when smushing
2	Apply horizontal smushing rule 2 when smushing
4	Apply horizontal smushing rule 3 when smushing
8	Apply horizontal smushing rule 4 when smushing
16	Apply horizontal smushing rule 5 when smushing
32	Apply horizontal smushing rule 6 when smushing
64	Horizontal fitting (kerning) by default
128	Horizontal smushing by default (Overrides 64)
256	Apply vertical smushing rule 1 when smushing
512	Apply vertical smushing rule 2 when smushing
1024	Apply vertical smushing rule 3 when smushing
2048	Apply vertical smushing rule 4 when smushing
4096	Apply vertical smushing rule 5 when smushing

8192	Vertical fitting by default
16384	Vertical smushing by default (Overrides 8192)

When no smushing rules are included in Full_Layout for a given axis, the meaning is that universal smushing shall occur, either by default or when requested.

Old_Layout: (Legal values -1 to 63)

-1	Full-width layout by default
0	Horizontal fitting (kerning) layout by default*
1	Apply horizontal smushing rule 1 by default
2	Apply horizontal smushing rule 2 by default
4	Apply horizontal smushing rule 3 by default
8	Apply horizontal smushing rule 4 by default
16	Apply horizontal smushing rule 5 by default
32	Apply horizontal smushing rule 6 by default

* When Full_Layout indicates UNIVERSAL smushing as a horizontal default (i.e., when none of the code values of horizontal smushing rules are included and code value 128 is included in Full_Layout) Old_Layout must be set to 0 (zero). Older FIGdrivers which cannot read the Full_Layout parameter are also incapable of universal smushing. Therefore they would be directed to the "next best thing", which is horizontal fitting (kerning).

NOTE: You should NOT add the -1 value to any positive code value for Old_Layout. This would be a logical contradiction.

See [Standardized Capabilities of current and future FIGdrivers](#) for the behavior of a FIGdriver when the Full_Layout parameter is absent (presumably in an older FIGfont).

The following rules establish consistency between Old_Layout and Full_Layout.

```

If full width is to be the horizontal default:
    Old_Layout must be -1.
    Full_Layout must NOT include code values 64 nor 128.

If horizontal fitting (kerning) is to be default:
    Old_Layout must be 0.
    Full_Layout must include code value 64.
    Full_Layout must NOT include code value 128.

If CONTROLLED smushing is to be the horizontal default:
    Old_Layout must be a positive number, represented by the added
    code values of all desired horizontal smushing rules.
    Full_Layout must include the code values for the SAME set of
    horizontal smushing rules as included in Old_Layout.
    Full_Layout must include code value 128.

If UNIVERSAL smushing is to be the horizontal default:
    Old_Layout must be 0.
    Full_Layout must include code value 128.
    Full_Layout must NOT include any code value under 64.

```

In general terms, if Old_Layout specifies horizontal smushing rules, Full_Layout must specify the same set of horizontal rules, and both must indicate the same horizontal default layout mode.

Setting Layout Parameters Step-by-Step

The following step-by-step process will yield correct and consistent values for the two layout parameters. You may skip this if you find the explanations above easier to use.

Step 1: Start with 0 for both numbers.

```

Write "Old_Layout" and "Full_Layout" on a piece of paper.
Write the number 0 next to each.
The number 0 may be crossed out and changed several times below.
Go to step 2.

```

Step 2: Set the DEFAULT HORIZONTAL LAYOUT MODE.

```

If you want to use FULL WIDTH as the default
    Make Old_Layout -1
    Go to step 3.
If you want to use HORIZONTAL FITTING (kerning) as the default
    Make Full_Layout 64
    Go to step 3.
If you want to use HORIZONTAL SMUSHING as the default

```

Make Full_Layout 128
Go to step 3.

Step 3: Specify HOW TO SMUSH HORIZONTALLY WHEN SMUSHING.

If you want to use UNIVERSAL smushing for the horizontal axis
Go to step 4.
If you want to use CONTROLLED smushing for the horizontal axis
Add together the code values for all the horizontal smushing
rules you want from the list below to get the horizontal
smushing rules total.

EQUAL CHARACTER SMUSHING	1
UNDERSCORE SMUSHING	2
HIERARCHY SMUSHING	4
OPPOSITE PAIR SMUSHING	8
BIG X SMUSHING	16
HARDBLANK SMUSHING	32

Horizontal smushing rules total: ____

If Full_Layout is currently 128
Change Old_Layout to the horizontal smushing rules total.
Increase Full_Layout by the horizontal smushing rules total.
Go to Step 4.
If Full_Layout is currently 0 or 64
Increase Full_Layout by the horizontal smusing rules total.
Go to Step 4.

Step 4: Set the DEFAULT VERTICAL LAYOUT MODE.

If you want to use FULL HEIGHT as the default
Go to step 5.
If you want to use VERTICAL FITTING as the default
Increase Full_Layout by 8192.
Go to step 5.
If you want to use VERTICAL SMUSHING as the default
Increase Full_Layout by 16384.
Go to step 5.

Step 5: Specify HOW TO SMUSH VERTICALLY WHEN SMUSHING.

If you want to use UNIVERSAL smushing for the vertical axis
Go to step 6.
If you want to use CONTROLLED smushing for the vertical axis
Add together the code values for all the vertical smushing
rules you want from the list below to get the vertical
smushing rules total.

EQUAL CHARACTER SMUSHING	256
UNDERSCORE SMUSHING	512
HIERARCHY SMUSHING	1024
HORIZONTAL LINE SMUSHING	2048
VERTICAL LINE SUPERSMUSHING	4096

Vertical smushing rules total: ____

Increase Full_Layout by the vertical smushing rules total.
Go to step 6.

Step 6: You're done.

The resulting value of Old_Layout will be a number from -1 to 63.
The resulting value of Full_Layout will be a number from 0 and 32767.

FIGfont Comments

After the header line are FIGfont comments. The comments can be as many lines as you like, but should at least include your name and Email address. Here is an example which also shows the header line.

```
flf2a$ 6 5 20 15 3 0 143
Example by Glenn Chappell <ggc@uiuc.edu> 8/94
Permission is hereby given to modify this font, as long as the
modifier's name is placed on a comment line.
```

Comments are not required, but they are appreciated. Please comment your FIGfonts.

Remember to adjust the Comment_Lines parameter as you add lines to your comments. Don't forget that blank lines DO count.

FIGcharacter Data

The FIGcharacter data begins on the next line after the comments and continues to the end of the file.

Basic Data Structure

The sub-characters in the file are given exactly as they should be output, with two exceptions:

1. Hardblanks should be the hardblank character specified in the header line, not a blank (space).
2. Every line has one or two endmark characters, whose column locations define the width of each FIGcharacter.

In most FIGfonts, the endmark character is either "@" or "#". The FIGdriver will eliminate the last block of consecutive equal characters from each line of sub-characters when the font is read in. By convention, the last line of a FIGcharacter has two endmarks, while all the rest have one. This makes it easy to see where FIGcharacters begin and end. No line should have more than two endmarks.

Below is an example of the first few FIGcharacters, taken from small.flf.

NOTE: The line drawn below consisting of "|" represents the left margin of your editor. It is NOT part of the FIGfont. Also note that hardblanks are represented as "\$" in this FIGfont, as would be described in the header line.

	\$@
	\$@
blank/space	\$@
	\$@
	\$@@
	_ @
	_ @
exclamation point	_ @
	(_) @
	_ @ @
	_ _ @
	() @
double quote	V V @
	\$ @
	_ @ @
	_ _ _ @
	_ _ . _ @
number sign	_ _ _ _ @
	_ _ _ _ @ @
	_ @
	_ _ @
dollar sign	(_ - < @
	/ _ / @
	_ _ @ @

Notice that each FIGcharacter occupies the same number of lines (6 lines, in this case), which must also be expressed in the header line as the Height parameter.

Also notice that for every FIGcharacter, there must be a consistent width (length) for each line once the endmarks are removed. To do otherwise would be an error.

Be aware of the vertical alignment of each FIGcharacter within its height, so that all FIGcharacters will be properly lined up when printed.

If one of the last sub-characters in a particular FIGcharacter is "@", you should use another character for the endmark in that FIGcharacter so that the intended "@" is not interpreted as an endmark. "#" is a common alternative.

Load a few existing FIGfonts into your favorite text editor for other examples.

Required FIGcharacters

Some FIGcharacters are required, and must be represented in a specific order. Specifically: all of the printable character codes from ASCII shown in the table below, in order, plus character codes 196, 214, 220, 228, 246, 252, and 223, in that order. In Latin-1, these extra 7 characters represent the following German characters: umlauted "A", "O", "U", "a", "o" and "u"; and also "ess-zed".

- Printable portion of the ASCII character set:

32 (blank/space)	64 @	96 `
33 !	65 A	97 a
34 "	66 B	98 b
35 #	67 C	99 c
36 \$	68 D	100 d
37 %	69 E	101 e
38 &	70 F	102 f
39 '	71 G	103 g

40 (72 H	104 h
41)	73 I	105 i
42 *	74 J	106 j
43 +	75 K	107 k
44 ,	76 L	108 l
45 -	77 M	109 m
46 .	78 N	110 n
47 /	79 O	111 o
48 0	80 P	112 p
49 1	81 Q	113 q
50 2	82 R	114 r
51 3	83 S	115 s
52 4	84 T	116 t
53 5	85 U	117 u
54 6	86 V	118 v
55 7	87 W	119 w
56 8	88 X	120 x
57 9	89 Y	121 y
58 :	90 Z	122 z
59 ;	91 [123 {
60 <	92 \	124
61 =	93]	125 }
62 >	94 ^	126 ~
63 ?	95 _	


- Additional required Deutsch FIGcharacters, in order:

```

196 (umlauted "A" -- two dots over letter "A")
214 (umlauted "O" -- two dots over letter "O")
220 (umlauted "U" -- two dots over letter "U")
228 (umlauted "a" -- two dots over letter "a")
246 (umlauted "o" -- two dots over letter "o")
252 (umlauted "u" -- two dots over letter "u")
223 ("ess-zed" -- see FIGcharacter illustration below)

```

Ess-zed >>--->



If you do not wish to define FIGcharacters for all of those required above, you MAY create "empty" FIGcharacters in their place by placing endmarks flush with the left margin. The Deutsch FIGcharacters are commonly created as empty. If your FIGfont includes only capital letters, please copy them to the appropriate lowercase locations, rather than leaving lowercase letters empty. A FIGfont which does not include at least all ASCII letters, a space, and a few basic punctuation marks will probably frustrate some users. (For example "@" is more frequently desired as a FIGcharacter than you may think, since Email addresses may be written as FIGures.)

Code Tagged FIGcharacters

After the required FIGcharacters, you may create FIGcharacters with any character code in the range of -2147483648 to +2147483647. (Over four billion possibilities, which is "virtual infinity" for this purpose.) One exception: character code -1 is NOT allowed for technical reasons. It is advisable to assign character codes such that the appearance of your FIGcharacters matches the expectations of ASCII/Latin-1/Unicode, with a few exceptions:

1. If a FIGcharacter with code 0 is present, it is treated specially. It is a FIGfont's "missing character". Whenever the FIGdriver is told to print a character which doesn't exist in the current FIGfont, it will print FIGcharacter 0. If there is no FIGcharacter 0, nothing will be printed.
2. If a FIGfont contains a non-Latin alphabet in character codes in the ASCII range 32-126 (which is discouraged), we have found it helpful to include a human-readable translation table as one of the FIGcharacters instead of a "glyph". Typically, the "~" would contain this table. The translation table FIGcharacter would contain a list of all the special characters in the FIGfont, along with the ASCII characters to which they correspond. Keep this table no more than 79 columns wide. (Thanks to Gedaliah Friedenberg for this idea.)
3. In more extensive Unicode fonts, you can assign a negative character code (other than -1) to one or more translation tables, similar to #2 above. (All Unicode character codes are positive.) And, you will most likely suggest within the comments that a user access one of several control files (See [Control Files](#) below) to gain access to Latin-2, Latin-3, or other 8-bit standardized character sets. The control files may redirect the "~" character to one of the negative character codes so that the translation table would display the table when "~" is given for input. Doing this allows you to still have a "~" FIGcharacter for those who do not use a control file.

Those FIGcharacters which are not required must have an explicit character code in a separate line preceding them, called a "code tag". A code tag contains the value of the character code, followed by whitespace (a few spaces), and perhaps an optional comment. The comment is usually the name of the FIGcharacter. The Unicode Consortium has assigned formal names to all officially accepted characters, and these may be used. An entire code tag, including the comment, should not occupy more than 95 columns. (Over 100 characters here may make older versions of FIGlet crash.)

Here is an example, showing two code tagged FIGcharacters after the last two required Deutsch FIGcharacters. Again, the line drawn below consisting of "|" represents the left margin of your editor, and is NOT part of the FIGfont.

[illegible]

A character code may be expressed in decimal (as shown above, numbers we're all familiar with), or in Octal (seldom used) or in hexadecimal.

Character codes expressed in octal must be preceded by "0" (zero), and if negative, "-" (minus) must precede the "0". There are eight octal digits: 01234567. You may recall octal numbers from school as "base 8 numbers".

Character codes expressed in hexadecimal must be preceded by "0x" or "0X". (That's also a zero.) If negative, the "-" must precede the "0x". There are 16 hexadecimal digits: 0123456789ABCDEF. (The "letter-digits" may also be lowercase.) Hexadecimal is "base 16".

It is common to express character codes less than 256 (in the range of an 8-bit character set) as decimal, while FIGfonts which extend into the Unicode range would have character codes expressed in hexadecimal. This is because the Unicode Standard expresses character codes in hexadecimal, which is helpful for programmers.

The code tagged FIGcharacters may be listed in any order, but simple sequential order is recommended.

If two or more FIGcharacters have the same character code, the last one in the FIGfont is the one used. It is common for the Deutsch FIGcharacters to be given twice in a FIGfont, just to maintain a consistent order for the Latin-1 range (128 to 255).

It is not advisable to assign character codes in the range of 1 to 31, since this range includes control characters in ASCII. Character code 127 is a delete in ASCII, and is also not advised. Character codes 128 to 159 are additional control characters in Latin-1, and they too should not be used. All of the above are legal, technically, but are not part of what is legal for input, so they could only be accessed by use of a control file. (See [Control Files](#) below.) If you are still tempted to use them, consider negative character codes instead, which are meaningless in all standardized character sets.

Again, the character code -1 is illegal for technical reasons.

Notes - Avoiding Errors and General Advice

It is very important that every character in a font has the same height, and, once the endmarks are removed, that all the lines constituting a single FIGcharacter have the same length. Be careful also that no lines in the font file have trailing blanks (spaces), as the FIGdriver will take these to be the endmarks. (FIGWin 1.0 will not consider blanks to be endmarks.)

Errors in a FIGfont can be detected by using the "chkfont" program, part of the standard FIGlet package.

For FIGWin users, the FIGWin program will report errors when a FIGfont is read in; it is less forgiving than FIGlet, which can produce nonsense if the FIGfont is incorrectly formatted.

Remember that sub-characters outside of the ASCII range will not necessarily display the same way on your system as on others.

The blank (space) FIGcharacter should usually consist of one or two columns of hardblanks and nothing else; slanted fonts are an exception to this rule. If the space FIGcharacter does not contain any hardblanks, it will disappear when horizontal fitting (kerning) or smushing occurs.

Again, if you design a FIGfont, please let us know!

Control Files

A FIGfont control file is a separate text file, associated with one or more FIGfonts, that indicates how to map input characters into FIGfont character codes. By default, FIGdrivers read single bytes from the input source and interpret them as Latin-1 FIGcharacters.

FIGlet version 2.2 (and later) can optionally interpret its input as DBCS or UTF-8 characters, making it possible to access FIGcharacters with codes outside the Latin-1 range (greater than 255).

In addition, though, all versions of FIGlet can use control files to transform specific character codes (or ranges of codes) as other codes (or ranges). Multiple control files can be specified, in which case multiple stages of transformation are performed.

The filename of a control file always ends with ".flc".

Standard Format

Control files contain several kinds of lines. Lines beginning with "#", as well as blank lines, are comment lines and are ignored. All other lines are command lines, with one of the following formats:

```
t inchar outchar
t inchar1-inchar2 outchar1-outchar2
number number
f
h
j
b
u
g{0|1|2|3} {94|96|94x94} [char]
g{L|R} {0|1|2|3}
```

where "inchar", "outchar", and "char" are either Latin-1 characters representing their own codes, or else are numeric character codes preceded by a "\" character; and "number" is a numeric character code with no preceding "\" character.

Thus "A" represents the code 65, as does "\65", and "\0x100" represents the code 256 (100 in hexadecimal). In addition, "\" (backslash followed by a space) represents the code 32 (space), and the following backslash sequences are also understood:

\a	code 7 (a bell/alert)
\b	code 8 (a backspace)
\e	code 27 (an ESC character)
\f	code 12 (a form feed)
\n	code 10 (a newline/line feed)
\r	code 13 (a carriage return)
\t	code 9 (a horizontal tab)
\v	code 11 (a vertical tab)
\\	code 92 (a backslash)

All of these combinations except perhaps "\\" are very unlikely to be used, but they are provided just in case they are needed.

Whitespace characters are used between "t" and "inchar" and between "inchar" and "outchar", but not around the "-" characters used in the second type of "t" command.

The term "string" refers to any number of characters represented in the format given above. The characters begin after the whitespace following the letter "s", and continue to the end of the line.

Anything following the first letter of an "f", "h", "j", or "u" command is ignored.

The first type of "t" command transforms characters with the code "inchar" into characters with the code "outchar". The second type of "t" command transforms characters in the range "inchar1" to "inchar2" as the corresponding codes in the range "outchar1" to "outchar2". Both ranges must be of the same size. The form "number number" is equivalent to a "t" command of the first type, and is provided for compatibility with the mapping tables issued by the Unicode Consortium.

Multiple transformation stages can be encoded in a single control file by using "f" commands to separate the stages.

Versions of FIGlet before 2.1 required that the first line of a control file consist of the signature string "flc2a". This signature line is still permitted in FIGlet 2.2 and later versions, but is no longer required.

Here is an example of a control file. The blanks at the beginning of each line are for readability only, and are not part of the file.

The following control file:

```
flc2a
t # $
t A-Z a-z
```

will map the "#" character to "\$", and will also convert uppercase ASCII to lowercase ASCII.

If a number of consecutive "t" commands are given, then for each character processed, only the first applicable command (if any) will be executed. Consider this control file:

```
t A B
t B A
```

It will swap the characters "A" and "B". If the FIGdriver reads an "A", the first command will change "A" to "B", in which case the second will not be executed. If the FIGdriver reads a "B", the first command will have no effect, and the second command will change "B" to "A". Here is another control file:

```
t A B
t A C
```

In this example, the second line is never executed. In short, a sequence of "t" lines "does what it ought to".

More complex files, in which a single character is acted upon by several "t" commands, can be set up using an "f" command. For example:

```
flc2a
t a-z A-Z
f
t Q ~
```

This control file specifies two transformation stages. In the first stage, lowercase ASCII letters are changed to their uppercase equivalents. The second stage maps any Q (whether original or a converted "q") into the "~" character. If the "f" command were omitted, "q" characters would remain "Q" and not be converted to "~".

Extended Commands

The "h", "j", "b", "u", and "g" commands are only understood by FIGlet version 2.2 or later. They control how a FIGdriver interprets bytes in the input. By default, the FIGdriver interprets each byte of input as a distinct character. This mode is suitable for most character encodings. All these commands are logically acted on before any other control file commands, no matter where in the sequence of control files they appear. They are also mutually exclusive; if more than one of these commands is found, only the last is acted on. Multiple "g" commands are permitted, however.

The "h" command forces the input to be interpreted in HZ mode, which is used for the HZ character encoding of Chinese text. In this mode, the sequence "~{" (which is removed from the input) signals that all following characters are two bytes long until the sequence "~}" is detected. In addition, the sequence "~~" is changed to just "~", and all other two-byte sequences beginning with "~" are removed from the input. The character code corresponding to a two-byte character is:

$$\text{first character} * 256 + \text{second character}$$

The "j" command forces the input to be interpreted in Shift-JIS mode (also called "MS-Kanji mode"). Input bytes in the ranges 128-159 and 224-239 are read as the high-order byte of a two-byte character; all other bytes are interpreted as one-byte characters. The value of a two-byte character is determined in the same way as in HZ mode.

The "b" command forces the input to be interpreted in DBCS mode, which is suitable for processing HZ or Shift-GB Chinese text or Korean text. Input bytes in the ranges 128-255 are read as the high-order byte of a two-byte character; all other bytes are interpreted as one-byte characters. The value of a two-byte character is determined in the same way as in HZ mode.

The "u" command forces the input to be interpreted in UTF-8 mode, which causes any input byte in the range 0x80 to 0xFF to be interpreted as the first byte of a multi-byte Unicode (ISO 10646) character. UTF-8 characters can be from 1 to 6 bytes long. An incorrectly formatted sequence is interpreted as the character 128 (normally an unused control character).

Otherwise, the input is allowed to contain ISO 2022 escape sequences, which are decoded to generate appropriate character codes. These character codes are *not* a subset of Unicode, but may be more useful in processing East Asian text. A brief explanation of ISO 2022 is given here in order to clarify how a FIGdriver should interpret it. The "g" command provides information for the ISO 2022 interpreter, and is explained below.

ISO 2022 text is specified using a mixture of registered character sets. At any time, up to four character sets may be available. Character sets have one of three sizes: single-byte character sets with 94 characters (e.g. ASCII), single-byte character sets with 96 characters (e.g. the top halves of ISO Latin-1 to Latin-5), or double-byte character sets with 94 x 94 characters (e.g. JIS 0208X-1983). Each registered character set has a standard designating byte in the range 48 to

125; the bytes are unique within character set sizes, but may be reused across sizes. For example, byte 66 designates the 94-character set ASCII, the 96-character set ISO Latin-2 (top half), and the 94 x 94 Japanese character set JIS 0208X-1983. In this document, the designating byte of a character set will be represented by <D>.

The four available character sets are labeled G0, G1, G2, and G3. Initially, G0 is the 94-character set ASCII, and G1 is the 96-character set ISO Latin-1 (top half). The other character sets are unassigned. The following escape sequences (where ESC = the byte 27) specify changes to the available character sets:

ESC (<D>	Set G0 to the 94-character set <D>
ESC) <D>	Set G1 to the 94-character set <D>
ESC * <D>	Set G2 to the 94-character set <D>
ESC + <D>	Set G3 to the 94-character set <D>
ESC - <D>	Set G1 to the 96-character set <D>
ESC . <D>	Set G2 to the 96-character set <D>
ESC / <D>	Set G3 to the 96-character set <D>
ESC \$ <D>	Set G0 to the 94 x 94 character set <D>
ESC \$ (<D>	Set G0 to the 94 x 94 character set <D>
ESC \$) <D>	Set G1 to the 94 x 94 character set <D>
ESC \$ * <D>	Set G2 to the 94 x 94 character set <D>
ESC \$ + <D>	Set G3 to the 94 x 94 character set <D>

Note that G0 may not be a 96-character set, and that there are two ways to specify a 94 x 94 character set in G0, of which the first is deprecated.

ISO 2022 decoding affects input bytes in the ranges 33 to 126 and 160 to 255, known as "the left half" and "the right half" respectively. All other bytes, unless they belong to a control sequence shown in this document, remain unchanged. Initially, the left half is interpreted as character set G0, and the right half as character set G1. This can be changed by the following control sequences:

SI (byte 15)	Interpret the left half as G1 characters
SO (byte 14)	Interpret the left half as G0 characters
ESC n	Interpret the left half as G2 characters
ESC o	Interpret the left half as G3 characters
ESC ~	Interpret the right half as G1 characters
ESC }	Interpret the right half as G2 characters
ESC	Interpret the right half as G3 characters
SS2 (byte 142)	Interpret next character only as G2
ESC N	Interpret next character only as G2
SS3 (byte 143)	Interpret next character only as G3
ESC 0	Interpret next character only as G3

This rich schema may be used in various ways. In ISO-2022-JP, the Japanese flavor of ISO 2022, only the bytes 33-126 and the G0 character set is used, and escape sequences are used to switch between ASCII, ISO-646-JP (the Japanese national variant of ASCII), and JIS 0208X-1983. In other versions, the G1 character set has 94 x 94 size, and so any byte in the range 160-255 is automatically the first byte of a double-byte character.

FIGdrivers that support ISO 2022 do so in the following way. Each character *i* is decoded and assigned to a character set <D>.

```

If the character belongs to a 94-bit character set,
  then if its value exceeds 128, it is reduced by 128,
  and the value 65536 * <D> is added to it,
  unless <D> is 66 (ASCII).
If the character belongs to a 96-bit character set,
  then if its value is less than 128, it is increased by 128,
  and the value 65536 * <D> is added to it,
  unless <D> is 65 (ISO Latin-1).
If the character belongs to a 94 x 94 character set,
  then the value is the sum of:
    the first byte * 256,
    plus the second byte,
    plus the value 65536 * <D>.

```

Thus, the character code 65 ("A") in ASCII remains 65, the character code 196 in ISO Latin-1 ("A-umlaut") remains 196, the character code 65 (0x41) in ISO-646-JP (whose <D> is 74 = 0x4A) becomes 0x4A0041 = 4849729, and the two-byte sequence 33 33 (0x21 0x21) in JIS 0208X-1983 (whose <D> is 65 = 0x41) becomes 0x412121 = 4268321. These codes may be used in compiling FIGfonts suitable for use with ISO 2022 encoded text.

The initial settings of G0 through G3 and their assignments to the left half and the right half can be altered in a control file by using "g" commands, as follows:

```
g {0|1|2|3} {94|96|94x94} [<D>]
```

specifies that one of G0-G3 is a 94, 96, or 94x94 character set with designating character <D>. If no designating character is specified, then a <D> value of zero is assumed.

For example, the list of control commands:

```
g 0 94 B
g 1 96 A
```

sets the G0 character set to ASCII (94-character set "B") and the G1 character set to the top half of Latin-1 (96-character set "A"). (This is the default setting).

To change the initial assignments of G0 to the left half and G1 to the right half, "g" commands of the form

```
g {L|R} {0|1|2|3}
```

For example, the command:

```
g R 2
```

causes right-half bytes (in the range 160-255) to be interpreted as G2. Whether these bytes are interpreted singly or in pairs depends on the type of character set that is currently available as G2.

Spaces may be freely used or omitted in "g" commands.

The standard FIGlet distribution contains mapping tables for Latin-2 (ISO 8859-2), Latin-3 (ISO 8859-3), Latin-4 (ISO 8859-4), and Latin-5 (ISO 8859-9). They can be used with the font "standard.flf", which contains all the characters used in these standards.

Standardized Capabilities of current and future FIGdrivers

We assert the following as the "Law" of our intentions:

Profit

All future FIGdrivers shall be FREE OF CHARGE to the general public via the Internet. Any advertisements of other works by the author must be in documentation only, and limited to ONE "screenful", and shall not appear by normal program behavior, nor interfere with normal behavior. No FIGdriver shall disable itself after a set period of time or request "donations". No FIGdriver shall offer any other FIGdriver with improved capability for creating FIGures in exchange for money.

Required Features of future versions

Future FIGdrivers must read and process FIGfont files as described in this document, but are not necessarily expected to process control files, smush, perform fitting or kerning, perform vertical operations, or even produce multiple lines in output FIGures.

FIGdriver NAMES

Future FIGdrivers must be named to include capitalized "FIG" and shall have an incremental version number specific to its own platform.

Backwards Compatibility of future versions

Any future FIGdriver created for the same platform as an existing FIGdriver, and using the same name as the existing FIGdriver, shall be considered a new version of the preceding FIGdriver, and shall contain all historical comments of updates to past versions on the same platform, and shall have full capability of the preceding versions. If the source code is not provided to the general public, it shall be at least provided to any potential developers of later versions, and such comments relating to past versions shall be accessible to any user by other means or documentation. If a new program is created on a platform that already has an existing FIGdriver, it must be given a new and distinct name. This allows multiple FIGdrivers to exist for the same platform with different capabilities.

The format of FIGfonts may not be modified to be non-backwards compatible UNLESS:

1. The new format is easily editable as an ASCII text file, beginning with the characters "flf" followed by a sequential number.
2. At least all of the same information can be derived from the new format as the prior format (currently "flf2"). This includes the main comments which give credit to the FIGfont designer.
3. Individuals are found who are willing and have the ability to either port or develop versions for at least UNIX, DOS, Windows, and Amiga which will read both the new formats AND the prior format (currently "flf2"), and retain the capability of past versions. It is intended that this will be expanded to include Macintosh if a GUI version exists. This list of required operating systems may be reduced if an operating system falls out of popularity or increased if a new operating system for which there is a FIGdriver comes into greater popularity, according to the consensus of opinions of past developers for the most popular operating systems.
4. A C, Java, or other version must always exist which can receive input and instructions either from a command line, a file, or directly over the internet so that FIGures can be obtained from internet-based services without the need to

download any FIGdriver.

5. All existing FIGfonts available from the "official" point of distribution (<http://st-www.cs.uiuc.edu/users/chai/figlet.html>), must be converted to the new format, and offered for download alongsidethe new versions.

The Function of Word Wrapping

All future FIGdrivers should duplicate these behaviors, unless a version is only capable of outputting one-line FIGures, which is acceptable as long no preceding versions exist for its platform which can output multiple-line FIGures.

FIGdrivers which perform word wrapping do so by watching for blanks (spaces) in input text, making sure that the FIGure is no more wide than the maximum width allowed.

Input text may also include linebreaks, so that a user may specify where lines begin or end instead of relying on the word wrapping of the FIGdriver. (Linebreaks are represented by different bytes on different platforms, so each FIGdriver must watch for the appropriate linebreaks for its particular platform.)

When a FIGdriver word wraps and there are several consecutive blanks in input text where the wrapping occurred, the FIGdriver will disregard all blanks until the next non-blank input character is encountered. However, if blanks in input text immediately follow a linebreak, or if blanks are the first characters in the input text, the blanks will be "printed", moving any visible FIGcharacters which follow on the same output line to the right. Similarly, if an image is right-aligned, and blanks immediately precede linebreaks or the end of input text, a FIGdriver will move an entire line of output FIGcharacters to the left to make room for the blank FIGcharacters until the left margin is encountered. (If the print direction is right-to-left, everything stated in this paragraph is reversed.)

Word processing programs or text editors usually behave similarly in all regards to word wrapping.

General Intent for Cross-platform Portability

Currently, all versions of FIGlet are compiled from C code, while FIGWin 1.0 is written in Visual Basic. Over time it is intended that a later version of FIGWin will be created using a GUI C programming language, and that the FIGlet C code shall continue to be written to be easily "plugged in" to a GUI shell. It is preferable for developers of FIGdrivers for new platforms to use C or a GUI version of C, so that when the core rendering engine of FIGlet is updated, it will be portable to other platforms.

Control File Commands

New control file commands may be added to later versions of this standard. However, the commands "c", "d", and "s" are permanently reserved and may never be given a meaning.

File Compression

FIGfonts (and control files) are often quite long, especially if many FIGcharacters are included, or if the FIGcharacters are large. Therefore, some FIGdrivers (at present, only FIGlet version 2.2 or later) allow compressed FIGfonts and control files.

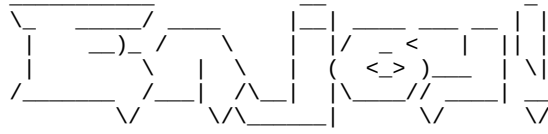
The standard for FIG compression is to place the FIGfont or control file into a ZIP archive. ZIP archives can be created by the proprietary program PKZIP on DOS and Windows platforms, or by the free program Info-ZIP ZIP on almost all platforms. More information on ZIP can be obtained at <http://www.cdrom.com/pub/infozip/Info-Zip.html>.

The ZIP archive must contain only a single file. Any files in the archive after the first are ignored by FIGdrivers. In addition, the standard extension ".zip" of the archive must be changed to ".flf" or ".flc" as appropriate. It does not matter what the name of the file within the archive is.

Chart of Capabilities of FIGlet 2.2 and FIGWin 1.0

The following chart lists all capabilities which are either new with the release of both FIGdrivers, or is not a common capability among both.

	FIGlet 2.2	FIGWin 1.0
Interpreting the Full_Layout parameter:	Yes	Yes
Universal smushing:	Yes	Yes
Supporting multi-byte input text formats:	Yes	No
Processing control files:	Yes	No
Changing default smushing rules:	Yes	No
Bundled with a GUI editor of FIGfonts:	No	Yes
Vertical fitting and smushing:	No	Yes



(Converted [original document](#) from txt to HTML by Markus Gebhard 2008)