

Second digit	First digit															
↓	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			space	0	@	P	`	p	Ä	ê	†	∞	¿	-		
1		⌘	!	1	A	Q	a	q	Å	ë	°	±	¡	—		
2		✓	"	2	B	R	b	r	Ç	í	‡	≤	¬	“		
3		◆	#	3	C	S	c	s	É	ì	£	≥	√	”		
4		Ⓐ	\$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘		
5			%	5	E	U	e	u	Ö	ï	●	µ	≈	’		
6			&	6	F	V	f	v	Ü	ñ	¶	∂	Δ	ˆ		
7			'	7	G	W	g	w	á	ó	ß	Σ	«	◇		
8			(	8	H	X	h	x	à	ò	®	Π	»	ÿ		
9			)	9	I	Y	i	y	â	ô	©	π	...			
A			*	:	J	Z	j	z	ä	ö	™	∫	⌵			
B			+	;	K	[	k	{	ã	õ	´	æ	À			
C			,	<	L	\	l		å	ú	¨	ø	Ã			
D			-	=	M	]	m	}	ç	ù	≠	Ω	Õ			
E			.	>	N	^	n	~	é	û	Æ	æ	Œ			
F			/	?	O	_	o	·	è	ü	Ø	ø	œ			

⌵ stands for a nonbreaking space, the same width as a digit.  
 The first four characters are only in the system font (Chicago).  
 The shaded characters are not in all fonts.  
 Codes \$D9 through \$FF are reserved for future expansion.

## Appendix A Transfer Modes

There are eight different pen modes. These modes are used to derive the color of a pixel when it is being drawn to. Each pixel is made up of a series of bits. The pen operates on the individual bits in a pixel as single units. In this way logical binary operations are well defined.

The codes for the various pen modes are different from the codes used in QuickDraw on the Macintosh. Similar modes for text, pen and pixel transfers all use the same codes. The codes for the inverted modes are the same as the original mode except that the high bit of the word is set.

The following transfer modes are available. (Each 1 and 0 is the value of a bit in a pixel.)

Mode           \$0000 (COPY)  
                  \$8000 (notCOPY)

Copy SRC (or not SRC) to destination. Copy is the typical drawing mode. For text, the fully colored text pixels (both foreground and background) are copied into the destination.

copy		Pen			Pen	
		0 1			0 1	
Dest.	0	0 1		0	1 0	
	1	0 1		1	1 0	

Mode           \$0001 (OR)  
                  \$8001 (notOR)

Overlay (OR) SRC (or not SRC) and destination. You can use this mode to non-destructively overlay new images on top of existing images and its inverse to overlay inverted images. For text, the fully colored text pixels (both foreground and background) are ORed with the destination.

OR		Pen			Pen	
		0 1			0 1	
Dest.	0	0 1		0	1 0	
	1	1 1		1	1 1	

**Mode**            **\$0002 (XOR)**  
**\$8002 (notXOR)**

Exclusive or (XOR) pen with destination. You can use this mode and its inversion for cursor drawing and rubber-banding. If an image is drawn in penXOR mode, the appearance of the destination at the image location can be restored merely by drawing the image again in penXOR mode. For text, the fully colored text pixels (both foreground and background) are XORed with the destination.

XOR		Pen		Pen		Pen		Pen
Dest.	0		0	1	0	1	0	1
	0		0	1	notXOR	0		1
	1		1	0	Dest.	0		0
						1		1

**Mode**            **\$0003 (BIC)**  
**\$8003 (notBIC)**

Bit Clear (BIC) pen with destination ((NOT pen) AND destination). You can use this mode to explicitly erase (turn off) pixels, often prior to overlaying another image. notBIC can be used to display the intersection of two images. For text, the fully colored text pixels (both foreground and background) are BICed with the destination.

BIC		Pen		Pen		Pen		Pen
Dest.	0		0	0	0	1	0	1
	0		0	0	notBIC	0		0
	1		1	0	Dest.	0		0
						1		1

**Special Text Modes.** The following modes are only used for text. They apply when drawing from a 1-bit per pixel world to a 2 or 4 bit per pixel world. This only occurs when drawing from the font to a destination pixel map.

**Mode**           **\$0004 (foreCOPY)**  
                  **\$8004 (notforeCOPY)**

Copies only the foreground pixels into the destination. Background pixels are not altered. The inverted mode inverts the foreground pixels before copying them.

**Mode**           **\$0005 (foreOR)**  
                  **\$8005 (notforeOR)**

ORs only the foreground pixels into the destination. Background pixels are not altered. The inverted mode inverts the foreground pixels before the OR operation occurs.

**Mode**           **\$0006 (foreXOR)**  
                  **\$8006 (notforeXOR)**

XORs only the foreground pixels into the destination. Background pixels are not altered. The inverted mode inverts the foreground pixels before the XOR operation occurs.

**Mode**           **\$0007 (foreBIC)**  
                  **\$8007 (notforeBIC)**

BICs only the foreground pixels into the destination. Background pixels are not altered. The inverted mode inverts the foreground pixels before the BIC operation occurs.

## Appendix B Hardware Summary

The Super Hi-Res Graphics hardware can display 200 scan lines and many colors. The following four features are controlled independently for each scan line:

Color Table	One of 16
Fill Mode	On or Off
Interrupt	On or Off
Color Mode	320 vs 640 pixels per scan line

The scan line control byte (SCB) controls these four features for each scan line. The low nibble of the SCB identifies the color table to be used for this scan line. Bit 4 is reserved. Bit 5 of the SCB controls fill mode: 1 is on, 0 is off. Bit 6 of the SCB controls interrupts: if the bit is set then an interrupt will be generated when the scan line is refreshed. Bit 7 of the SCB controls the mode: 0 is 320, 1 is 640.



### Color Table

A color table is a table of 16 two byte entries. The low nibble of the low byte is the intensity of the color blue. The high nibble of the low byte is the intensity of the color green. The low nibble of the high byte is the intensity of the color red. The high nibble of the high byte is not used. Pixels in 320 mode are 4 bits wide and their numeric representation identifies a color in the color table. Pixels in 640 mode are two bits wide and their numeric representation identifies a color in a subset of the full color table. The first pixel in the byte (bits 0 and 1) selects one of four colors in the table from 0 thru 3. The second pixel in the byte (bits 2 and 3) selects one of four colors in the table from 4 thru 7. The third pixel in the byte (bits 4 and 5) selects one of four colors in the table from 8 thru 11. The fourth pixel in the byte (bits 6 and 7) selects one of four colors in the table from 12 thru 15.

HighByte		LowByte	
High Nibble	Low Nibble	High Nibble	Low Nibble
Reserved	Red	Green	Blue

### **Fill Mode**

When fill mode is active, the 0th color in the color table becomes inactive. A pixel with a numeric value of zero serves as a place holder indicating that the pixel should be displayed as the same color last displayed.

#### **Scan Line Values**

1 0 0 0 0 2 0 0 0 0 0 1 0 0 0 0

#### **Colors Shown**

B B B B B W W W W W B B B B B

### **Interrupts**

Interrupts can be used to synchronize drawing with vertical blanking so pixels are not changed as they are being drawn (a pixel is drawn once every 1/60 of a second). Interrupts can also be used to change the color table before a screen is completely drawn. This will allow a program to show more than 256 colors on the screen at once (but at the cost of servicing the interrupt).

## Appendix C Comparison to QuickDraw

QuickDraw has the following functional parts. Next to each part I indicate where the part will fall.

Environmental Control	Core
Rectangle Drawing	Core
Line Drawing	Core
Pixel Map Transfer	Core
Region Clipping	Core
Cursor Support and Drawing	Core
Utilities	Core
Text	Core
Region Manipulation	Core
Round Things (ovals, circles, round rects and arcs)	RAM
Pictures	RAM
Polygons	Core
Advanced Routines	Core

Each routine in QuickDraw is listed below with its corresponding QuickDraw II routine. An entry under QuickDraw II of "same" or "similar" means that the routine will work just like or somewhat like the corresponding QuickDraw routine. A minus sign indicates that the routine will not be present. Some entries are the names of calls as they will appear in QuickDraw II (different from QuickDraw). A question mark indicates that we are not yet sure. Finally, An explanation point indicates that prototype code is running today.

Mac QuickDraw Routine	QuickDraw II Core	QuickDraw II RAM
InitGraf	Different!	
OpenPort	same!	
InitPort	same!	
ClosePort	same!	
SetPort	same!	
GetPort	same!	
GrafDevice	-	
SetPortBits	SetPortLoc!	
PortSize	same!	
MovePortTo	same!	
SetOrigin	same!	
SetClip	same!	
GetClip	same!	
ClipRect	same!	
BackPat	same!	
InitCursor	same!	
SetCursor	same!	

HideCursor	same!
ShowCursor	same!
ObscureCursor	same!
HidePen	same!
ShowPen	same!
GetPen	same!
GetPenState	same!
SetPenState	same!
PenSize	same!
PenMode	same!
PenPat	same!
PenNormal	same!
MoveTo	same!
Move	same!
LineTo	same!
Line	same!
SetFont	SetFont!
TextFace	sortof!
TextMode	same!
TextSize	no
SpaceExtra	same!
DrawChar	same!
DrawString	same!
DrawText	same!
CharWidth	same!
StringWidth	same!
TextWidth	same!
GetFontInfo	same!
ForeColor	SetForeColor!
BackColor	SetBackColor!
	GetForeColor!
	GetBackColor!
ColorBit	-
SetRect	same!
OffsetRect	same!
InsetRect	same!
SectRect	same!
UnionRect	same!
PtInRect	same!
Pt2Rect	same!
PtToAngle	?
EqualRect	same!
EmptyRect	same!
FrameRect	same!
PaintRect	same!
EraseRect	same!
InvertRect	same!
FillRect	same!



FrameOval		same!
PaintOval		same!
EraseOval		same!
InvertOval		same!
FillOval		same!
FrameRoundRect		same!
PaintRoundRect		same!
EraseRoundRect		same!
InvertRoundRect		same!
FillRoundRect		same!
FrameArc		same!
PaintArc		same!
EraseArc		same!
InverArc		same!
FillArc		same!
NewRgn	same!	
DisposeRgn	same!	
CopyRgn	same!	
SetEMptyRgn	same!	
SetRectRgn	same!	
RectRgn	same!	
OpenRgn	same!	
CloseRgn	same!	
OffsetRegion	same!	
InsetRgn	same!	
SectRgn	same!	
UnionRgn	same!	
DiffRgn	same!	
XorRgn	same!	
PtInRgn	same!	
RectInRgn	same!	
EqualRgn	same!	
EmptyRgn	same!	
FrameRgn	same!	
PaintRgn	same!	
EraseRgn	same!	
InvertRgn	same!	
FillRgn	same!	
ScrollRect	same!	
CopyBits	PaintPixels!	
OpenPicture		same
PicComment		same
ClosePicture		same
DrawPicture		same
KillPicture		same
OpenPoly	same!	
ClosePoly	same!	

KillPoly	same!
OffsetPoly	same!
FramePoly	same!
PaintPoly	same!
ErasePoly	same!
InvertPoly	same!
FillPoly	same!
AddPt	same!
SubPt	same!
SetPt	same!
EqualPt	same!
LocalToGlobal	same!
GlobalToLocal	same!
Random	same!
GetPixel	similar!
StuffHex	unlikely
ScalePt	same!
MapPt	same!
MapRect	same!
MapRgn	same!
MapPoly	same!
SetStdProcs	same!
StdText	similar!
StdLine	similar!
StdRect	similar!
StdRRect	similar!
StdOval	similar!
StdArc	similar!
StdPoly	similar!
StdRgn	similar!
StdBits	similar!
StdComment	similar!
StdTxMeas	similar!
StdGetPic	similar!
StdPutPic	similar!

## Appendix D To QuickDraw II ERS

### Fonts And Text In QuickDraw II:

### To Boldly Write Where No Person Has Writ Before

Version 00.01

August 12, 1986

Bennet Marks

**nraiu etoain shrdliuu etoain shrdliu etoain shrdliu etoain**  
**! shrdliu etoain shrdliuu etoain shrdliu e**  
hrdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu et  
dlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu etoain shr  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu et  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu et  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu et  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu eto  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu eto  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu eto  
rdlu etoain shrdliuu etoain shrdliu etoain shrdliu etoain shrdliu et

**Fonts And Text In QuickDraw II**

**Fonts And Text In QuickDraw II**  
**Revision History**

<b>Version 00.00</b>	<b>July 15, 1986</b>	<b>Bennet Marks</b>
<b>.First Draft</b>		
<b>Version 00.01</b>	<b>August 12, 1986</b>	<b>Bennet Marks</b>
<b>fontID field of Cortland Font Record renamed family</b>		

# Fonts And Text In QuickDraw II

## Table Of Contents

1. Introduction	D-1
1.1 About This Document	D-1
1.2 Abstract	D-1
2. Font Definition	D-2
2.1 Overview	D-2
2.2 The Cortland Font Definition	D-3
2.3 The Cortland Header Fields	D-4
2.4 The MF Part Of A Cortland Font	D-5
3. Characters, Fonts, And Drawing	D-6
3.1 Characters	D-6
3.2 Fonts	D-9
3.2.1 The Font Rectangle	D-9
3.2.2 Other Fields In The Font	D-11
3.2.3 The Font Strike	D-12
3.2.4 Defined Vs. Undefined Characters	D-13
3.2.5 The Location Table	D-14
3.2.6 The Offset/Width Table	D-15
3.3 Character Backgrounds And The Font Bounds Rectangle	D-17
3.3.1 Character Backgrounds	D-17
3.3.2 The Font Bounds Rectangle	D-18
3.4 Drawing, And The Text Buffer	D-20

## Fonts And Text In QuickDraw II

4. Controlling Text Display	D-21
4.1 Character Spacing:	D-21
SetCharExtra, GetCharExtra, SetSpaceExtra, GetSpaceExtra	
4.2 Style Modifications:	D-22
SetTextFace, GetTextFace	
4.2.1 Bolding	D-22
4.2.2 Underlining	D-22
4.3 Other Options:	D-23
SetFontFlags, GetFontFlags	
5. Discussion Of The Main Calls	D-23
5.1 Text Drawing Calls:	D-23
DrawChar, DrawText, DrawString, DrawCString	
5.2 Text Width Calls:	D-24
CharWidth, TextWidth, StringWidth, CStringWidth	
5.3 Text Bounds Calls:	D-24
CharBounds, TextBounds, StringBounds, CStringBounds	
5.4 Managing The Text Buffer:	D-25
SetBufStuff, ForceBufStuff, SaveBufDims, RestoreBufDims	
5.4.1 The Whole Text Buffer Catalog	D-26
5.4.2 Sizing The Buffers	D-27
5.4.3 Saving And Restoring The Buffer Sizes	D-29
5.5 Font Information:	D-30
GetFontInfo, GetFontGlobals, GetFGSize	

# Fonts And Text In QuickDraw II

by Bennet Marks

## 1. Introduction

### 1.1 About This Document

This document contains a detailed description of the handling of text and fonts in QuickDraw II, including the definition of a Cortland font. It is intended as a supplement to the QuickDraw II ERS. Most application writers, most of the time, will not need any more information than is included in that ERS. But if you are designing a font, writing a font editor, making use of unusual fonts or an usually large variety of fonts, or otherwise mucking about in the seemingly foggy areas of QD II text display, this document may be helpful. More casual readers may want to browse it, to see if it answers any questions they may have lurking in the backs of their minds, or reveals any powerful or secret functionality which may turn out to be useful. Or not . . . .

This represents the current state of the world, as of QuickDraw II Version 01.01. Any discrepancies from V.01.00 (which was never well-documented anyway) will be noted.

Some conventions: (1) references to any field of the Cortland font record are boldfaced. (2) the word "strings" is often used to mean "strings, text blocks, or Cstrings". This should be clear from context.

Criticisms, complements, questions, and comments to Bennet Marks, MS 22-X, x6245. Thanks.

### 1.2 Abstract

In our treatment of text drawing and text measurement, we have stayed

## Fonts And Text In QuickDraw II

very close to the Macintosh model. The Cortland font definition is very similar to Macintosh's; a simple conversion algorithm allows us to use any font developed for the Mac. Most Macintosh QuickDraw text calls are duplicated precisely in QuickDraw II. Any differences are due to the following:

- (1) we have added some information to the beginning of the font definition;
- (2) we do not have resources, so we are not planning on a full-blown Font Manager in the Macintosh style. This required some changes in the QD II calls;
- (3) we have added some calls - notably "bounding box" calls (TextBounds and its siblings) - that are missing from Macintosh QuickDraw;
- (4) we have added calls - DrawCString, CStringWidth, and the like - to handle the CString data type (a sequence of characters terminated by a 0 byte);
- (5) some features (such as scaling, italicizing, shadowing, and outlining text) were not implemented in our ROM. They may be added in RAM latter.

References: see Inside Macintosh, chapters 6 and 7, and Macintosh Revealed, chap. 8

## 2. Font Definition

### 2.1 Overview

A Cortland font consists of a variable-length header, followed by a Macintosh font record (we'll refer to this embedded Mac font as the MF part of the Cortland font).

The header is included to compensate for Cortland's lack of resources and a complete Font Manager; it is of variable length to allow us to add extra information at a later date.



## Fonts And Text In QuickDraw II

The MF part is exactly like a true Macintosh font, except for one thing - the "gender" of integers. Mac's 68000 (as well as other processors in that family) stores integers with the high byte first (i.e., high byte at lower memory location); Cortland's 65816 stores them with low byte first. So in converting a Mac font to a Cortland font, we must swap the upper and lower byte of each integer. This does not apply to the font strike (bitImage), which can be used as is.

### 2.2 The Cortland Font Definition

**CortFontRec =**  
**RECORD**

**offsetToMF :** INTEGER; {offset in words to Mac Font part}  
**family :** INTEGER; {font family number - previously fontID}  
**style :** INTEGER; {style font was designed with}  
**size :** INTEGER; {point size}  
**version :** INTEGER; {QD II version number}  
**fbrExtent:** INTEGER; {font bounds rectangle extent}

{additional fields may be present here}

{Mac font part follows:}

**fontType :** INTEGER; {font type}  
**firstChar :** INTEGER; {ASCII code of first defined character}  
**lastChar :** INTEGER; {ASCII code of last defined character}  
**widMax :** INTEGER; {maximum character width}  
**kernMax :** INTEGER; {maximum leftward kern}  
**nDescent :** INTEGER; {negative of descent}  
**fRectWidth :** INTEGER; {width of font rectangle}  
**fRectHeight :** INTEGER; {font height}  
**owTLoc :** INTEGER; {offset in words to offset/width table}  
**ascent :** INTEGER; {font ascent}  
**descent :** INTEGER; {font descent}  
**leading :** INTEGER; {leading}  
**rowWords :** INTEGER; {width of font strike in words}

**{bitImage :** ARRAY[1..rowWords,1..fRectHeight] of WORD;  
    {font strike}

## Fonts And Text In QuickDraw II

**{locTable :        ARRAY[firstChar. . lastChar+2] of INTEGER;**  
**{location table}**

**{owTable :         ARRAY[firstChar. . lastChar+2] of INTEGER;**  
**{offset/width table}**  
**END;**

### 2.3 The Cortland Font Header Fields

We have included some information about the font in the header. Fonts designed at a later time may include additional information, which could be utilized by later versions of QD II; this is why the header is of variable length. For upward and downward compatibility of QD II and Cortland fonts, two fields are particularly useful:

**offsetToMF** - this is the offset, in words, from this field to the Macintosh font (MF) part included in the Cortland font (specifically, to the fontType field). The header is therefore  $2 * \text{offsetToMF}$  bytes long. In QD II, V.01.01, **offsetToMF** = 6; the header is 12 bytes. Future fonts may have longer headers, containing font information that can be utilized by future versions of QD II. To assure that these improved fonts can be used by older versions of QD II, the **offsetToMF** field provides a reliable jump over this extra font information to the start of the Mac part of the font. Of course, an older QD II will not be able to make use of new header fields added since it was implemented; but at least it will be able to find the information it can use.


**version** - this is the version number of the QD II for which the font was designed. Later versions of QD II may, as mentioned, be able to utilize extra information which is included in later fonts; by checking this field, they can avoid trying to find and use information not included in an older font. (Presumably a newer QD II, alerted by the version number to the lack of such information, would use some default or calculated values.)

Examples of extra information that may be included in later fonts and used by later versions of QD II are: thickness of underline, slope of italicized letters, "smearing extent" of boldface, and the like (in Mac, these are determined by the Font Manager).

The other header fields are:

## Fonts And Text In QuickDraw II

**family** - an integer identifying the font, irrespective of size or style. This can be thought of as corresponding to the font's name - Courier, Geneva, etc. Presumably we will use the same numbers as Mac. More detail to follow at some later date.

 In previous versions of this document, this field was called **fontID**. This conflicted with the grafport field **fontID**, a longword that contains this field, the font style, and the point size. The grafport won.

**style** - this indicates the style the font was designed with; in other words, the style that the font "thinks" it is. Application writers and/or graphic designers may design "pre-italicized" fonts, bolded fonts, or the like, for reasons of aesthetics or time performance. When QD II is asked to apply a certain style when drawing a character or string, it first checks this field. If the field indicates that the requested style is already part of the font, the drawing call will not apply the styling algorithm. This prevents, say, pre-italicized fonts from getting re-italicized. (See 4.2 for definition of style word.)

**size** - this is the point size of the font.

**fbrExtent** - font bounds rectangle extent. The maximum horizontal distance, in pixels, from the character origin to any foreground or background pixel of any character in the font. This field was added to QuickDraw II as of V.01.01. (see 3.3.2)

### 2.4 The MF Part Of A Cortland Font

A Mac font, or in this case the Macintosh font part of a Cortland font, consists of 4 sections:

(1) a fixed-length record containing general information, such as the font height, the maximum character width, etc.

(2) the font strike (named **bitImage** in the font record definition), which is a pixelmap containing the image of every character defined in the font, strung one after another. The pixelmap is in 1 bit/pixel form. Its width, measured in words, is given by the **rowWords** field of the font record; its

## Fonts And Text In QuickDraw II

height, measured in pixels, is given by the `fRectHeight`.

(3) the location table (`locTable`), an array of integers which indicates for each defined character where its image in the font strike begins.

(4) the offset/width table (`owTable`), an array of integers. For each character, the low byte of its entry in the offset/width table (the "character offset") indicates how the character image to be drawn should be positioned with respect to the current pen location (using a somewhat arcane encoding which will be detailed later - see 3.2.6); the high byte indicates how far the pen should be advanced after the character is drawn ("character width").

This table is also used to identify a font's "missing characters", that is, characters not defined in the font. An `owTable` value of -1 (`$FFFF`) marks a missing character, which must be handled specially by the text calls. More on this below (see 3.2.4).

A detailed description of the meanings and uses of these various fields and arrays will be given below, with a general discussion of characters, fonts, and drawing.

### 3. Characters, Fonts, And Drawing

This section largely duplicates similar chapters of Inside Macintosh and Macintosh Revealed. Some differences of the Cortland scheme are presented, and some information that was unclear to me in my first few (actually, several) readings of the Macintosh documentation is emphasized. It is devoutly to be hoped that this will be helpful to those who come after me (and I mean that in the most non-violent sense of the phrase).

#### 3.1 Characters

A character image is a rectangular array of bits, representing pixels. The "on" (1) bits are called the character foreground pixels.

By convention, no "padding" is included on the left or the right of the character image; that is, neither the left nor right column is blank (all 0's). (When characters are actually drawn, the space between them is

## Fonts And Text In QuickDraw II

determined by information contained in the font tables and elsewhere; this is discussed below.)

There is no such convention for the top and bottom rows of a character image; they may be blank.

The number of rows in the character image (including any blank rows) is called the character height. Every character in a font must have the same height (which is why blank top or bottom rows are sometimes necessary). This is called the font height, or `fRectHeight` in the font record.

The number of columns in a character image is called the character image width, or just the image width. Note that a character can have an image width of 0. For example, the space has a 0 image width; its "character image" consists of no pixels at all. (The character height, in such a case, is determined by the font that the space character is part of.)

The character rectangle is a rectangle which encloses the character image. Its width is the image width of the character, which may vary from character to character in a font; its height is the character height, which is the same for all characters in a font.

Each character has a number associated with it called the character width, found in the offset/width table. This is the number of pixels the pen position is to be advanced after the character is drawn. This is different from the image width, and the distinction is very important. For example, the space character has 0 image width, but some positive character width, which determines the size of the space. Some characters have a non-zero image width but a 0 character width - one example is an umlaut, which is meant to be typed over a vowel. The umlaut is drawn first, then the vowel is drawn with the same pen location. Characters with 0 character width are called, somewhat morbidly, dead characters.

Also associated with every character in a font is its baseline and its character origin. The baseline is a horizontal line which separates the image into two sets of rows, one set above and one below. (Remember that in QuickDraw II, as in QuickDraw, horizontal and vertical lines fall between pixels, rather than running through them.) The position of the baseline depends on the font's ascent and descent fields; it is chosen so that there are ascent rows above it and descent rows below. The

## Fonts And Text In QuickDraw II

baseline will be in the same horizontal position for every character in the font. Any foreground pixels of a character image that lie below the baseline are collectively called the character's descender. Most characters don't have a descender, but, in an average font, characters like "q" and "y" do.

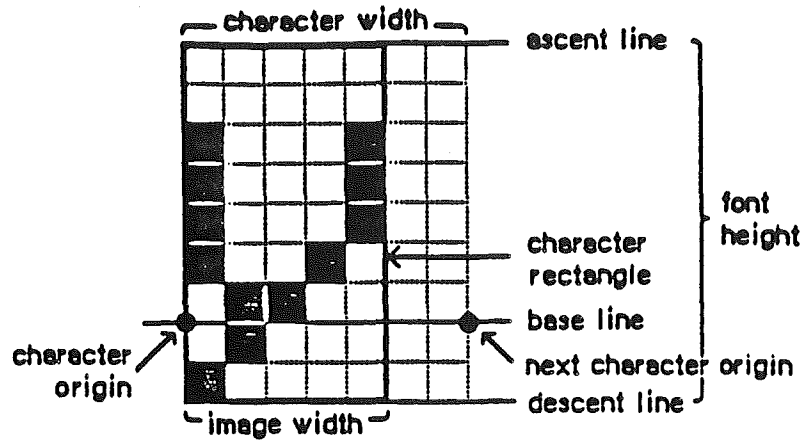
The ascent line is the horizontal line just above the top row of a character, and the descent line is the line just below the bottom row. They will also be the same for every character in the font.

For each character, its character origin is a point on the baseline which is used to position the character for drawing. This point may be between pixels of the character image, to the right of them, or to the left. (Here, note that points lie between pixels, not on them.) Its location relative to the character image can be calculated by the character offset in the offset/width table, as will be detailed later (3.2.6 again). When the character is drawn, it is placed in the destination pixelmap so that its character origin coincides with the current pen location.

For many letters, the character origin is located on the left edge of the character image, so that, when the character is drawn, its leftmost foreground pixels fall just to the right of the pen. Sometimes the character origin is between pixels of the character image (or, rarely and perversely, entirely to the right of the image). When such a character is drawn, some of its pixels will fall to the left of the pen position. This is called kerning (to the left). In such a case, the distance in pixels from the character origin to the left edge of the character is called the character's leftward kern.

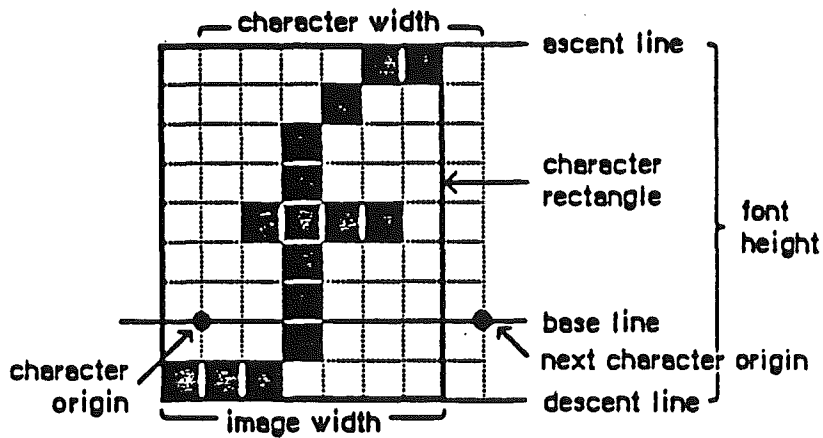
When character image pixels fall to the right of the new pen position after the character is drawn, the character is said to kern to the right. For various reasons this is not nearly so significant or interesting as kerning to the left. Kerning in either direction can cause letters to overlap each other. (See Figures 1 and 2 on next page.)

## Fonts And Text In QuickDraw II



**character - no kerning**

**Figure 1**



**character - kerns left**

**Figure 2**

### 3.2 Fonts

#### 3.2.1 The Font Rectangle

Imagine all the defined characters of a font drawn so that their character origins coincide. The result would be a black mess of foreground pixels.

## Fonts And Text In QuickDraw II

The smallest rectangle completely enclosing this mess is called the font rectangle. (This is different from the font bounds rectangle, defined in 3.3.2.)

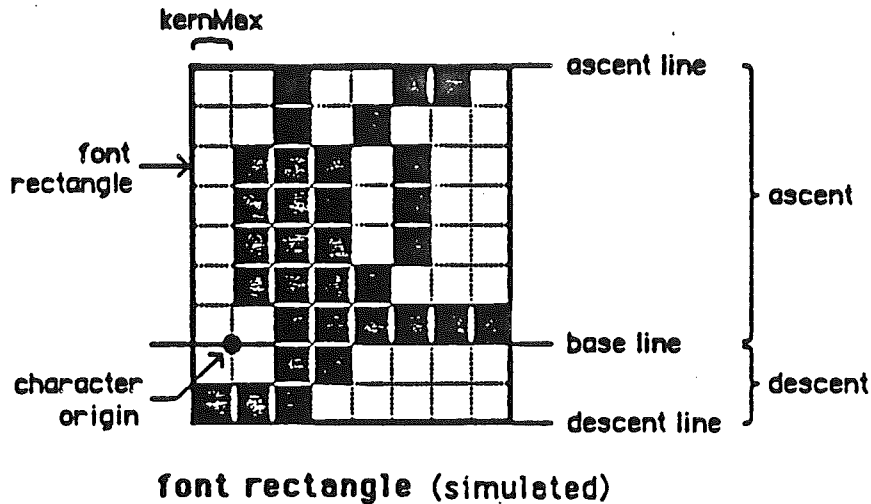


Figure 3

Several fields of the font measure aspects of the font rectangle:

**kernMax** - this is the distance in pixels from the font rectangle's (common) character origin to the left edge of the font rectangle. If the left edge of the font rectangle is to the left of the character origin - that is, if any character in the font actually kerns to the left - **kernMax** is represented as a negative number. If the character origin lies on the left edge of the font rectangle, **kernMax** is 0.

[Every Picture Tells A Story, Sometimes Fictional - Part 1: it is difficult to read the "sense" of a quantity represented by a distance in a picture. In Figure 3, **kernMax** = -1; but **ascent** = +7 and **descent** = +2]

That takes care of most fonts. However, it is quite possible that in some fonts the left edge of the font rectangle is 1 or more pixels to the right of the character origin. In this case, it seems appropriate - at least from a mathematician's point of view - to assign **kernMax** a positive value, even though this bends the terminology a bit; people do not usually say of a character that, for example, leaves two columns of blank pixels between the pen position and its image that it "kerns to the left" 2 pixels, or -2 pixels, or anything at all. (Not that the subject comes up at most dinner



## Fonts And Text In QuickDraw II

parties, anyway.) We will adopt this convention, permitting positive values for **kernMax**, at least for now.

**fRectWidth** - the width in pixels of the font rectangle. Note that this may be more than the maximum character image width, because the font rectangle's left and right extremes may come from different characters.

**fRectHeight** - the height in pixels of the font rectangle.

**ascent** - the number of pixel rows above the (common) baseline in the font rectangle (or, in any character).

**descent** - the number of pixel rows below the baseline in the font rectangle (or, in any character). Note **fRectHeight = ascent + descent**.

**nDescent** - negative of descent.

(An obscure aside: for typical fonts - those in which the font rectangle at least touches its character origin - **ascent** and **descent** will be non-negative, and **kernMax** and **nDescent** will be non-positive. But strange fonts can be imagined - or, worst yet, designed - in which these restrictions can be dropped. Ideally, QuickDraw II will handle these just as well. I may document these cases later, for the curious and/or obsessive.)

Look for the return of the font rectangle when we get into the offset/width table of the font! Coming soon to a raster near you! Just when you thought it was safe to go back into the drawing space again . . . .

### 3.2.2 Other fields In The font

**fontType** - this is left over from Mac, where \$9000 indicates a proportional font and \$B000 a fixed-width font. QD II ignores this field.

**firstChar** - ASCII code of the first defined character in the font. See 3.2.4.

**lastChar** - ASCII code of the last defined character of the font. See 3.2.4.

**widMax** - the maximum character width (pen displacement) of any

## Fonts And Text In QuickDraw II

character in the font, measured in pixels.

**owTLoc** - the offset, in words, from this field to the font offset/width table (**owTable**).

By adding  $2 * \text{owTLoc}$  to the memory address of this field, you get a pointer to the **owTable**. In order to get a pointer to the **locTable**, you must subtract  $2 * (\text{lastChar} - \text{firstChar} + 3)$  from the **owTable** pointer. There's no **locTLoc** field in the font record. (Why not? Don't ask me, I only work here.)

**leading** - the recommended number of blank pixel rows between the descent row of one line of text and the ascent row of the next. Applications may use this or not, as they please.

**rowWords** - the width of the font strike, in words. See below.

### 3.2.3 The Font Strike

The font strike (called **bitImage** in the font definition) is a 1 bit/pixel pixelmap consisting of the character images of every defined character in the font, placed sequentially in order of increasing ASCII code from **firstChar** to **lastChar+1** (we'll get to that "+1" business in a minute - see 3.2.5). The character images in the font strike abut each other; no blank columns are left between them. Since all the characters of a font have the same height, the font strike is just one long pixelmap with no jumps or undefined stretches, and with a height of **fRectHeight**. The strike is padded on the right, if necessary, with enough extra pixels (on each row) to make the row width a multiple of 16, that is, an integral number of words. This width, measured in words, is found in the **rowWords** field of the font record. (See Figure 4 on next page.)

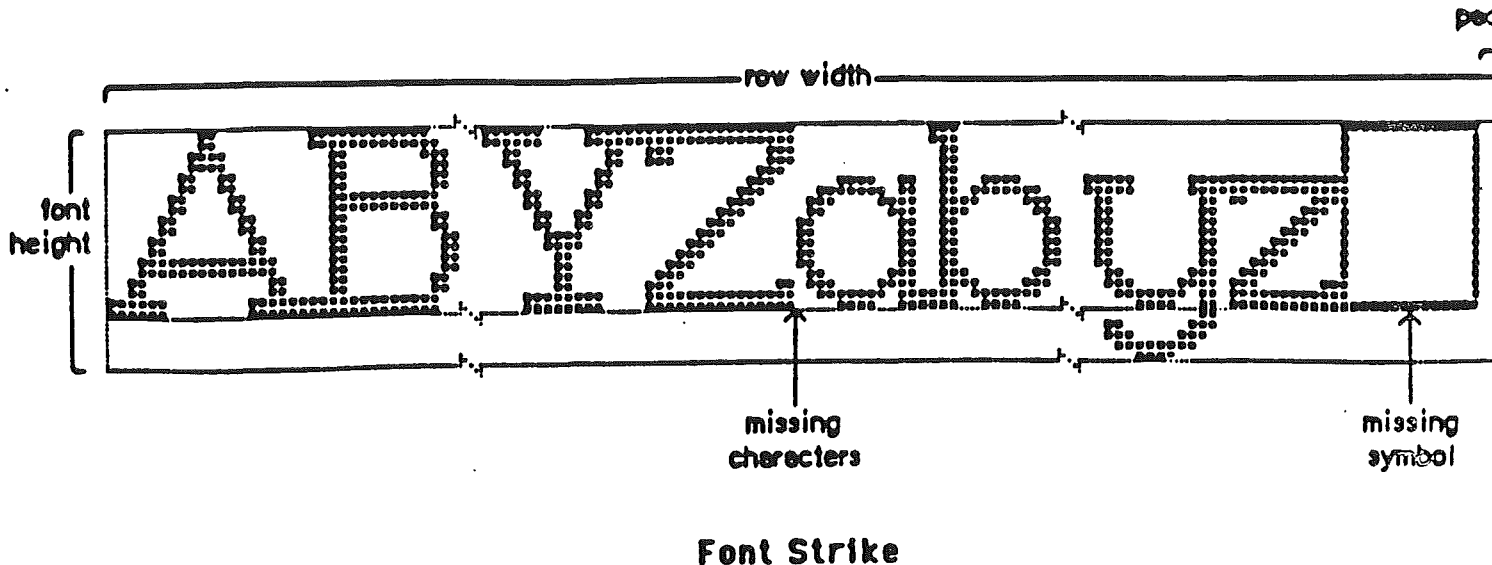


Figure 4

### 3.2.4 Defined Vs. Undefined Characters

Not every possible ASCII code must have a character image in the font strike. The font may leave some characters undefined; these are called missing characters (although "undefined characters" is good enough for me). Every character with a code less than firstChar or greater than lastChar+1 is undefined. There may be other undefined characters as well. The offset/width table (owTable) has an entry for every code from firstChar to lastChar+2, inclusive (we're up to "+2" already, and I haven't even told you about "+1"?!? Patience, patience.). If a character's entry in the offset/width table is -1 (\$FFFF), then the character is undefined ("missing").

Character code lastChar+1 is a special case. Immediately following lastChar in the font strike is a character known as the missing symbol, which is to be used in place of any missing character (it is exactly because of the confusion of these two phrases that I prefer "undefined character"). This character must be present in the font strike. It has entries in the locTable and the owTable, and its entry in the owTable must not be -1. For all purposes the missing symbol is a defined character with ASCII code lastChar+1. In many fonts the missing symbol is a hollow rectangle; in the current ROM system font, it's a white-on-black

question mark.

Whenever the QD II text-handling routines encounter a missing character - less than `firstChar`, greater than `lastChar+1`, or having an `owTable` entry of -1 - they immediately substitute the missing symbol for the character, using the missing symbol's character image, `locTable` entry, and `owTable` entry wherever needed.

### 3.2.5 The Location Table

The location table (`locTable`) is an array of integers with an entry for each character code from `firstChar` to `lastChar+2`. It is used to find character images in the font strike. For each defined character, its entry gives the distance in pixels from the beginning of the font strike to the beginning of the character's image in the font strike ("beginning", here, means left edge). This indicates where the character image starts. To see where it ends, take the next `locTable` entry (the beginning of the next character image), and subtract 1. Since the character images abut each other, this will give you the precise limits of the character image. The image width of a defined character with code `C` is `locTable[C+1] - locTable[C]`. This may be 0.

In order for this scheme to work, two conditions must hold:

(1) the `locTable` entry for an undefined character must be the same as the entry for the next defined character. This prevents undefined characters, which have no image in the strike, from interfering with the hunt for images of defined characters.

Note that there always will be a "next defined character", because the missing symbol, which serves as a defined character, is tacked on at the end of the strike.

(2) in order to get the character image for the missing symbol, there has to be an entry in the `locTable` following the missing symbol's. For this reason `locTable[lastChar+2]` is included, and is set equal to the length of the font strike in pixels, ignoring the "padding to a word boundary" that is added to the font strike. (Told you we'd get to `lastChar+2`!)

### 3.2.6 The Offset/Width Table

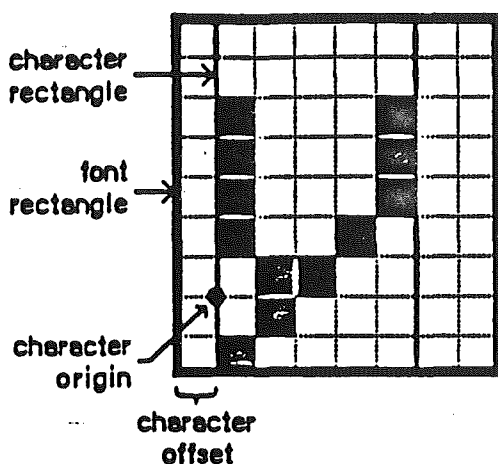
The offset/width table (`owTable`) is an array of integers with an entry for each character code from `firstChar` to `lastChar+2`. If a character's entry is -1, the character is undefined ("missing").

Otherwise the entry's low and high bytes are the character width and character offset, respectively. Both are interpreted as numbers in the range 0-254 (255 is ruled out to avoid the case where both bytes are 255, giving us an entry of -1, which would mark a missing character).

The character offset is used to calculate the position of character origin relative to the image, in the following way: the offset is added to the font's `kernMax`. The result is the (horizontal) distance in pixels from the character origin to the left edge of the image. If the result is negative, then the origin is to the right of the image's left edge (the character kerns leftward). If the result is positive, the origin is to the left of the image's left edge. (A result of 0 means that the character origin sits on the left edge of the image). Since we already know that the character origin must lie on the baseline (whose position is determined from `ascent` and `descent`), this locates the origin precisely.

If you draw the font rectangle, and look at a particular character's character rectangle within it, the character offset is seen to be the offset, in pixels, between the left edge of the font rectangle and the left edge of the character rectangle. Hence the name. (See Figure 5 on next page).

## Fonts And Text In QuickDraw II



**character rectangle in font rectangle**

**Figure 5**

The low byte of the offset/width table entry gives the character width, which is the distance in pixels the pen should be advanced (to the right - my apologies to the Hebrew scholars among us) after the character is drawn. In applications, this distance can be affected by a number of calls, particularly `SetCharExtra` and `SetSpaceExtra`. There is, however, a general rule in QD II, which can be simply stated as: if you're dead, you stay dead. Translation: any character whose character width (from the offset/width table, unmodified) is 0 will not have that width changed by `chExtra`, `spExtra`, style modifications, non-proportionality, or any other effect. We assume that characters are given 0 width only for some very good reason, and we're not going to mess with it.

**⚠ Caution:** any modification or combination of modifications that result in a character width of less than 0 or greater than 255 will wreak havoc with the drawing routines, and are not allowed.

This includes `chExtra`, `spExtra`, style mods, etc. QD II does not check for this condition. It's up to you. Good luck.

The `lastChar+2` entry of the offset/width table is set to -1.

### 3.3 Character Backgrounds And The Font Bounds Rectangles

#### 3.3.1 Character Backgrounds

As mentioned before, a character's foreground consists of all the 1 pixels in its image. You would think that the background consists of all the 0's in the image. Not quite. In QD II, we extend the background on the left to include any pixels to the left of the image's left edge, but to the right of the character origin (and between the ascent and descent lines). On the other side, we extend the background on the right to include any pixels (between ascent and descent) to the right of the image's right edge, but to the left of the character origin of the next character (that is, to the left of the new pen position). Any new pixels added in this way are considered background pixels.

(A clearer way to say this might be: the foreground of a character consists of all 1 bits in its character image. The background consists of all 0 bits in the image and all non-foreground pixels that are to the right of the character origin, to the left of the subsequent character origin [ $=$  character origin + character width], above the descent line, and below the ascent line.)

Of course, in some cases there is no extending to be done. If the character kerns to the left, then no left extension is necessary; if it kerns to the right, then no extension to the right is needed.

This is a very natural definition of background. If you're going to draw, say, a green character with a red background (tacky, except during Christmas), the red background will usually (that is, for characters that don't stretch entirely from the old pen position to the new) extend a little to the left and/or right of the character's image. This is what people generally want for a background. But, in addition, when characters do kern, the background extends as far left or right as the kerning, so the kerned part of the character doesn't jut out into never-never land.

This brings us to the definition of the character bounds rectangle: it is the smallest rectangle enclosing all the foreground and background pixels of a character. It may be somewhat larger than the character rectangle,

## Fonts And Text In QuickDraw II

which encloses the image, because the bounds rectangle takes into account the character width (pen positions) as well as the image width. The width of a character's bounds rectangle is called the character bounds width.

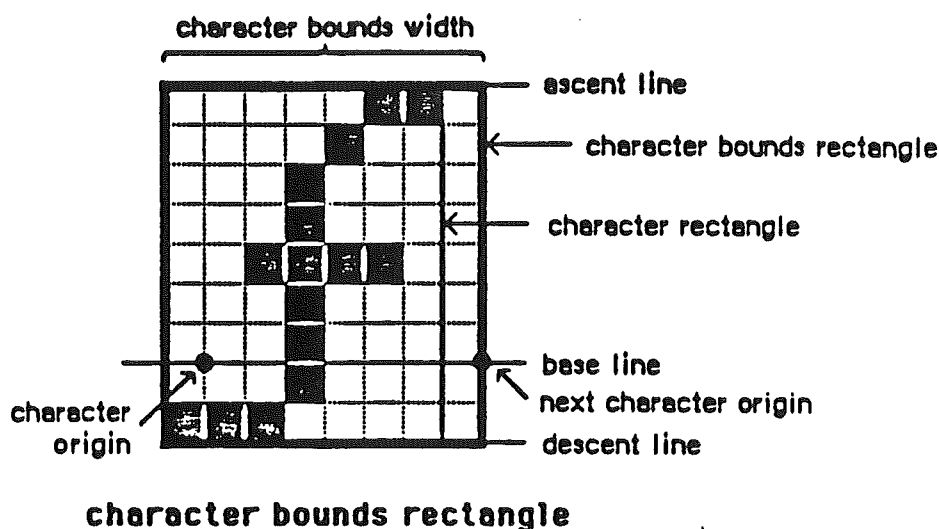


Figure 6

QD II includes calls for measuring character bounds rectangles, and corresponding routines for strings, Cstrings, and text.

### 3.3.2 The Font Bounds Rectangle

In order to get some new, useful measures for the "width" of a font, we define the font bounds rectangle. Imagine that, for all of a font's characters, the characters' bounds rectangles were drawn so that all the character origins coincided. The resulting rectangle (more precisely, the rectangle which is the union of all these rectangles) is called the font bounds rectangle. This rectangle includes all pixels, foreground and background, of every character in the font. (Consequently, it may be bigger than the font rectangle, which is only guaranteed to include all the foreground pixels.)

We define: fbrWidth to be the width of the font bounds rectangle; fbrRightExtent to be the distance from the (common) character origin to the right edge of the font bounds rectangle; and fbrLeftExtent to be the distance from the origin to the left edge (all distances measured in pixels, and as positive numbers). Finally, let fbrExtent be the maximum of



## Fonts And Text In QuickDraw II

`fbrLeftExtent` and `fbrRightExtent`. (`fbrWidth` plays no role right now, but seemed worth defining anyway.)

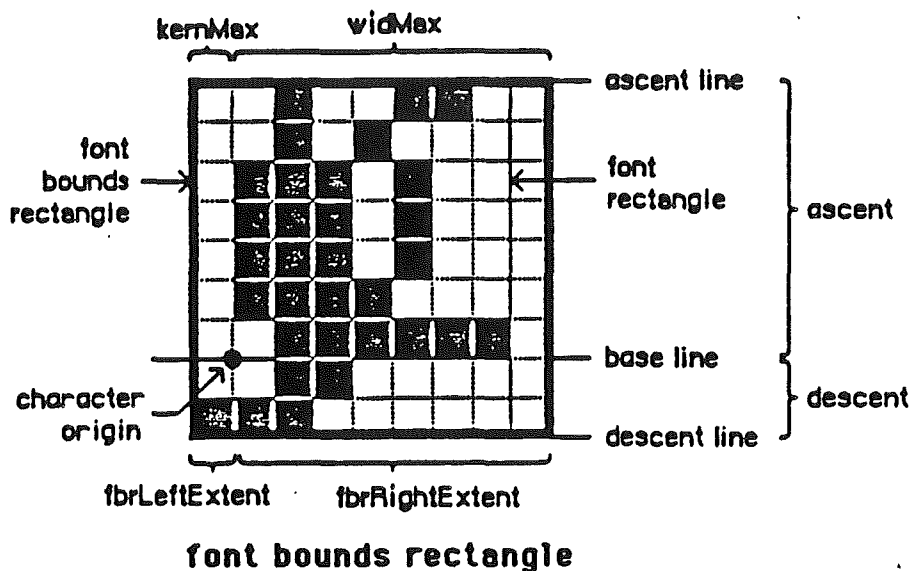


Figure 7

`fbrExtent` may seem like a fairly obscure number, but it has a natural interpretation. It is the furthest possible horizontal distance from the pen location to (the far edge of) any pixel that can be altered by drawing any character in the font. In many ways it is a more precise measure of the width of a font than `widMax` or `fRectWidth`.

[Every Picture Tells A Story, Sometimes Fictional - Part 2: it would seem from Figure 7 that `fbrLeftExtent` and `kernMax` are the same, or rather that `fbrLeftExtent` = `-kernMax`. This is true if and only if `kernMax` is zero or negative; if `kernMax` is positive (that is, if every character image starts at least 1 pixel to the right of the character origin) then `fbrLeftExtent` is zero. This makes `fbrLeftExtent` easy to calculate. It also looks like `fbrRightExtent` is the same as `widMax`; but if any character kerns to the right beyond the reach of `widMax`, then `fbrRightExtent` will be bigger than `widMax`. This makes `fbrRightExtent` a little more slippery.]

☞ The `fbrExtent` field was added to the Cortland font definition in QuickDraw II as of V.01.01. It's needed for the safe handling of the text buffer. It is not included in the Macintosh font definition. If you are converting a Mac font to the Cortland, `fbrExtent` can be calculated by

## Fonts And Text In QuickDraw II

using the CharBounds call on character codes 0-255 and doing some simple arithmetic (the CharBounds call itself doesn't need a valid value for fbrExtent, so it can be called for the calculation). This only has to be done once for each font.

### 3.4 Drawing, And The Text Buffer

Whenever a character or string is to be drawn, it is first drawn into the text buffer, a 1 bit/pixel pixelmap reserved for the private use of the QD II text-drawing calls. For strings, only those characters that have a chance of making it into the destination pixelmap (we will make this precise later; see 5.4.1) are actually drawn; the others, to the left and the right, only contribute to the cumulative pen displacement. So there is no reason for an application to try to "pre-clip" characters out of long strings, unless it has a uniquely fast way of doing so.

The text buffer starts out empty at the beginning of each drawing call. Successive characters of a string are drawn into it, with an internal "text buffer pen" incremented by the character width each time. Regardless of the ultimate text mode (txMode in the grafport), characters are drawn into the text buffer in OR mode. So characters that kern into each other do not interfere destructively. (For this reason, with certain text modes like SrcXor you can get different results if you put up a string one character at a time, using DrawChar, than if you put up the whole string with DrawString. In the DrawChar case, overlapping characters may cancel out some pixels.)

Once the character or string is safely in the text buffer, any requested style mods (underlining, bolding, etc.) are applied to it. Then the text buffer is transferred to the destination. Individual bits are replaced with 2 or 4 bits, depending on the "chunkiness" of the destination; the bit patterns used are the grafport's fgColor for the 1 (foreground) bits, bgColor for the 0 (background) bits. During the transfer, the text image is clipped to the current clipping region. The surviving pixels are combined with the destination's pixels, according to whatever text mode is in use. If and when the result makes it to the screen, the bit patterns will be translated into colored pixels according to the current color map(s).

## 4. Controlling Text Display

Various QD II calls affect text display. Generally they set some field of the current grafport which is used in the text-drawing process. Matching the "set" calls are corresponding "get" calls.

SetForeColor, GetForeColor, SetBackColor, GetBackColor, SetTextMode, and GetTextMode deal with the grafport fields fgColor, bgColor, and txMode, whose effects were described just above (3.4). The other display control calls deal with:

### 4.1 Character Spacing:

SetCharExtra, GetCharExtra, SetSpaceExtra, GetSpaceExtra

These calls set and/or get two fields in the grafport, chExtra and spExtra, which can alter the character widths (pen displacement) when characters are drawn. Each is a fixed-point number, with a one-word integer part and a one-word fractional part.

chExtra is added to the character width of each character as it is drawn, except for "dead characters" - as mentioned above, character widths of 0 are left that way.

Adding chExtra to a character width will, of course, give us character origin positions with a fractional part. During any text-drawing call, QD II keeps track of this fractional part and, when drawing a character, rounds its character origin position to the nearest integer (1/2 - that is, \$8000 - is rounded up). The fractional part is not remembered after the call is completed; each drawing call starts off fresh.

chExtra (which is not present in Mac) was included because some fonts that look fine in 320 mode appear too closely spaced in 640 mode; putting an extra pixel between letters seems to help in these cases.

spExtra works the same way, but it is only applied to the space character. It is commonly used to help in justifying text. Note that "space character", here, means ASCII \$20 and nothing else. In particular, the "non-breaking space" included in many fonts is unaffected by the spExtra

## Fonts And Text In QuickDraw II

field. `spExtra` is cumulative with `chExtra`.

These values are set by `SetCharExtra` and `SetSpaceExtra` (and can be fetched by `GetCharExtra` and `GetSpaceExtra`). In theory, the application can set `chExtra` and `spExtra` to any fixed-point value, even negative ones. However, as mentioned before, any values that cause a character to have a character width of less than 0 or greater than 255 pixels will cause no end of trouble. You have been warned!

### 4.2 Style Modifications:

`SetTextFace`, `GetTextFace`

These calls set and get the `txFace` field of the `grafport`, which determines the style to be applied to the text. In the ROM, only two bits are defined:

bit 0:	bold
bit 2:	underline

If one of these bits is set and the corresponding bit of the style field of the current font is not set, then the appropriate style modification is applied to the text (see style under 2.3). The next two sections describe the current implementation of the style mods.

#### 4.2.1 Bolding

As currently implemented, bolding has the following effect on characters and strings:

- (1) characters are spaced further apart: 1 pixel is added to every character width (except for dead characters).
- (2) each character is drawn twice, once in its expected position, then again 1 pixel to the right. The two images are ORed together. (This transformation is actually applied to the entire text buffer at once, after all the characters have been drawn into it. Saves time, that way.)

#### 4.2.2 Underlining

## Fonts And Text In QuickDraw II

If the font has a descent greater than or equal to 2, then a horizontal line, 1 pixel thick, is drawn 2 pixels under the baseline, extending the length of the character(s) to be drawn. This "length" takes into account both the starting and ending pen locations and any kerning to the left or the right. The underline is not necessarily continuous; it "shies away" from descenders. Specifically, a pixel of the line is omitted if there is a foreground pixel of the text image immediately above, below, to the right, or to the left of it.

If the font descent is less than 2, no underlining is done, regardless of the value of `txFace`.

When both bolding and underlining are called for, bolding is done first (it does make a difference).

### 4.3 Other Options:

#### SetFontFlags, GetFontFlags

The `fontFlags` field of the `grafport` is set by `SetFontFlags` and fetched by `GetFontFlags`. In the current ROM, only the last bit (bit 0) of this word is defined. If it is set, then the font is used as a non-proportional font - every character is given the same character width, namely `widMax` (the maximum character width field from the font definition). Well, almost every character - as usual, characters with character width 0 are exempted.

When non-proportionality is in effect, `chExtra`, `spExtra`, style modifications, etc., are applied to the character width after it has been set to `widMax`.

All other bits of this field are reserved, and should be set to 0.

## 5. Discussion Of The Main Calls

### 5.1 Text Drawing Calls:

`DrawChar`, `DrawText`, `DrawString`, `DrawCString`

## Fonts And Text In QuickDraw II

The specified character or string of characters is drawn, using all the current information - font, style, mode, etc. The current pen position is used as the character origin of the first character. The pen is advanced by the sum of the character widths. Note that, although the text image is clipped to the current clip region, the pen is not "clipped" in any way; the new pen position can be outside the current grafport bounds.

Near the edges of its drawing space ( $\pm 16K, \pm 16K$ ), QD II is unreliable; this applies to text drawing as well as to any other kind. Calls which would draw outside the space can cause catastrophic results. **Beware!**  
**Here there be dragons!**

### 5.2 Text Width Calls:

`CharWidth`, `TextWidth`, `StringWidth`, `CStringWidth`

These calls return the total pen displacement that would result if the character or sequence of characters were to be drawn. Nothing is actually drawn, however. The width calls take into account current styles, `chExtra`, `spExtra`, text flags, etc. But they do not take kerning (which is independent of pen displacement) into account; that's a job for the text bounds calls. Note that the width calls only return a pen displacement, not a new pen location. They make no use of the current pen location, and they don't change it.

### 5.3 Text Bounds Calls:

`CharBounds`, `TextBounds`, `StringBounds`, `CStringBounds`

These calls return the smallest rectangle that would enclose all the foreground and background pixels of the character or string ("or text block or `Cstring`") of characters if they were to be drawn, starting at the current pen location. The rectangle is given in the local coordinates of the current grafport.

Unlike the text width calls, these calls take kerning into account, as well as pen movement. The bounds rectangle extends to the left as far as the

## Fonts And Text In QuickDraw II

starting pen position or the leftmost kerning pixel (if any) of the text image, whichever is further to the left; similarly it extends as far right as the new pen position or the rightmost kerning pixel (ditto), whichever is further to the right. See 3.3, above, for an equally awkward description of this idea. But, at the least, the bounds rectangle is reliable; any pixel which might be changed by a text-drawing call is inside the corresponding bounds rectangle.


The rectangle extends up (from the current pen location) to the ascent line, and down (ditto) to the descent line. It is not clipped to any clipping region. It takes into account style mods, `chExtra`, `spExtra`, etc. Note that the bounds rectangle is not actually drawn by these calls; its coordinates are simply returned to the application.

Some strings ("or text or Cstrings"), or possibly even some characters, may have no foreground or background pixels. In the case of a character, it would have to have 0 image width and 0 character width - a space with no length. Strings may have zero length (no characters), or be composed entirely of the spaceless spaces just described. In these cases, the text bounds calls return a degenerate rectangle - specifically, one whose right and left edges are the same (namely, the current pen location's x coordinate). The upper and lower edges of the rectangle will be the ascent and descent line (relative to the pen's y coordinate), as usual.

Why these bounds calls were not included in Mac is beyond me.

### 5.4 Managing The Text Buffer:

`SetBufStuff`, `ForceBufStuff`, `SaveBufDims`, `RestoreBufDims`

 **Warning!** These calls affect the QD II clip buffer as well as the text buffer!

These are calls which affect the size of the text buffer and the way it is used. They also affect the QD II clip buffer; for details on that, see the QD II ERS.

Before detailing the calls and their parameters, we start with a general discussion of the text buffer:

### 5.4.1 The Whole Text Buffer Catalog

When a string (or text block or Cstring) is to be drawn into a pixelmap, it is first drawn into the text buffer. Characters of the string that fall far outside the destination's left or right boundaries are not actually drawn into the text buffer; only their character widths are used, to determine where the string actually enters the destination (on the right) and/or what the final pen location should be (on the left).

For the text-drawing calls to handle this safely and efficiently, QD II must have certain information about the largest pixelmap sizes and character sizes it will have to deal with. For one thing, the text buffer must be at least as wide (in pixels) as the widest destination pixelmap that may be used (actually, it must be a little wider, to avoid disaster when drawing characters that fall partly in and partly out of the destination); and it has to be as high as the highest font. For another thing, in order to decide if a pen location is so "far outside" the destination that a character drawn with that origin can't possibly impinge on the destination, QD II needs to know the width of the widest possible character. "Widest", here, includes not only image width and character width, but any elongations due to `chExtra`, `spExtra`, style modifications, etc. Any pixel that can be touched by a character's foreground or background must be considered.

This is what `fbrExtent` was created for. It describes how far away from the current pen location any "alterable" pixel can be. But `fbrExtent` depends only on the font, and does not take into account style mods and the like. This is why we have two calls: `SetBufStuff`, which provides (generous) defaults for any character elongations, and `ForceBufStuff`, which puts things more under the application's control.

**Important Note:** you may never need to call either of these routines! Please continue reading to see if you qualify. You may already be a winner . . . .

When `QDStartUp` is called, it creates a text buffer which is twice as high as the system font, wide enough to support the `MaxWidth` parameter of `QDStartUp`, and capable of handling characters twice as wide as the system font characters ("wide" in the sense of `fbrExtent`). It also



## Fonts And Text In QuickDraw II

permits the use, with any font, of any  $chExtra \leq fbrExtent$  (of that font);  $spExtra \leq fbrExtent$ ; and it allocates up to 36 extra pixels per character to accomodate style modifications (right now, all we have is bolding, which adds 1 pixel to a character; but we hope to have italics later, and italicizing a large font can stretch its horizontal extent quite a lot). If your application is only going to deal with fonts and text display parameters that fall within those limits, you can trust to the defaults and never call `SetBufStuff` or `ForceBufStuff`. Otherwise, read on:

### 5.4.2 Sizing The Buffers

`SetBufStuff` takes 3 parameters:

<code>MaxWidth:</code>	<code>INTEGER;</code>
<code>MaxFontHeight:</code>	<code>INTEGER;</code>
<code>MaxFBRExtent:</code>	<code>INTEGER;</code>

`MaxWidth` is the width in bytes (not pixels) of the largest pixelmap the application will draw into (a value of 0 indicates screen width). It will override the value supplied to `QDStartUp`. `MaxFontHeight` is the height, in pixels, of the tallest font the application will have to work with. `MaxFBRExtent` is the `fbrExtent` of the widest (i.e., greatest `fbrExtent`) font the application will work with. The call resizes the clip buffer and the text buffer to accomodate these sizes.

In addition, `SetBufStuff` "pads" the text buffer and signals the routine that determines if a character is "far outside" the destination to allow for: (1) values of `chExtra` and `spExtra`  $\leq$  the `fbrExtent` of the font in use at any given time; and (2) an extra 36 pixels of style modification added to the width of any character.

(It should now be obvious that `QDStartUp` makes an internal call to `SetBufStuff` with `MaxWidth` from the `QDStartUp` parameters, `MaxHeight` twice the size of the system font, and `MaxFBRExtent` twice the size of the system font `fbrExtent`.)

`SetBufStuff`'s three parameters are only used to size the text buffer (and the QD II clip buffer, but let's not worry about that now). When it comes time to actually draw a string, and QD II must decide which characters might make it into the destination and which ones don't have a snowball's chance, it uses the `fbrExtent` of the current font (which may be way

## Fonts And Text In QuickDraw II

smaller than MaxFBRExtent), the current values of spExtra, chExtra, txFace, etc., and, for a "destination pixelmap width", the width of the active portion of the current grafport's pixelmap (its minRect, to be specific). Therefore large values for SetBufStuff's parameters may soak up some memory for the text buffer size, but will not cost much in time lost drawing characters into the text buffer that will never make it into the destination. This also means that, once the text buffer is sized, the MaxFBRExtent parameter can be forgotten. (This is not true for ForceBufStuff.)

ForceBufStuff takes the same parameters as SetBufStuff and performs the same functions; however, it does not "pad" the text buffer at all. Any extra pixels that might be added to a character bounds width due to chExtra, spExtra, style mods, or whatever, should be added into the maxFBRExtent parameter by the application making the call.

ForceBufStuff, like SetBufStuff, sizes the buffer(s) on the basis of its parameters; and, when a string is actually drawn, only the width of the current grafport's pixelmap is considered, not all of MaxWidth. But, unlike SetBufStuff, ForceBufStuff forces QD II to use the MaxFBRExtent parameter to decide which characters are in and which out, rather than trying to calculate a "current" fbrExtent value. ForceBufStuff is for those times when you're going to do something weird, like a customized style modification, and QD II is not arrogant enough to think it can anticipate you. Consequently, when ForceBufStuff is called, its MaxFBRExtent value must be remembered for subsequent drawing calls. I mention all of this only because, in the SaveBufDims and RestoreBufDims, there is an asymmetry in the parameters handed back, depending on whether the text buffer was originally "set" (MaxFBRExtent no longer needed) or "forced" (MaxFBRExtent must be remembered). See 5.4.3.

It is of course permissible to call SetBufStuff or ForceBufStuff every time you change fonts, or even every time you call SetCharExtra, SetTextFace, or whatever. But this is not recommended. Sizing (and clearing) buffers can be quite time-consuming. The routines should probably be called once (if at all), with the maximum realistic values for each of the parameters, and never again. We may supply some RAM-based software tools for getting these values, maximized over all fonts in the system file.

## Fonts And Text In QuickDraw II

**SetBufStuff** and/or **ForceBufStuff** are useful if you are using enormous fonts, unreasonably large values of **chExtra** and **spExtra**, style mods that distend characters in some horrible way, or the like. However - surprise! - they can also be used to specify a text buffer which is smaller and more efficient than the default buffer provided by QD II. For example, if you plan on only using the system font, and no **chExtra**, **spExtra**, or style mods, then calling **ForceBufStuff** with the system font height and system **fbrExtent** will create a smaller, tighter text buffer which will save memory, save time used in buffer-clearing, and do a better job of identifying which characters don't need to be drawn into the text buffer (another time saving).

You know best what your application will need. But don't be stingy at the cost of flexibility. Also, realize that adding in a few "fudge factor" pixels to **MaxFBRExtent** may cost a tiny amount in space and time, but may save you disaster if you've, say, forgotten about bolding. Precise calculations of **MaxFBRExtent** for the **ForceBufStuff** call are not necessary; upper limits will do.

### 5.4.3 Saving And Restoring Buffer Sizes

**SaveBufDims** and **RestoreBufDims** are included for orderly "context-switching" between subprograms. **SaveBufDims** saves the "state" of the clip buffer and text buffer sizes in the form of an 8-byte record:

```
DimsRecord =  
RECORD  
MaxWidth:          INTEGER;  
TextBufHeight:     INTEGER;  
TextBufRowWords:   INTEGER;  
FontWidth:         INTEGER;  
END;
```

**MaxWidth** is the current value of the application-set maximum pixelmap width, in bytes. **TextBufHeight** is the current text buffer height, in pixels; and **TextBufRowWords** is the current width of the text buffer, in words (quite a varied collection of measurement units, isn't it?). **FontWidth** serves two purposes: if it is zero, it means the buffer was set up with a call to **SetBufStuff**; if it is non-zero, then the buffer was set up with a call to **ForceBufStuff**, and the value of **FontWidth** is equal to the

## Fonts And Text In QuickDraw II

MaxFBRExtent parameter used in that call.

RestoreBufDims restores the buffer dimensions on the basis of the DimRecord it is given.

Regardless of through what call the text buffer is sized or resized - by QDStartUp, SetBufDims, ForceBufDims, or RestoreBufDims - the application is not responsible for clearing it. The calls take care of clearing the text buffer automatically. Also note that SaveBufDims and RestoreBufDims do not save and restore the contents of the text buffer, only the parameters related to its size.

### 5.5 Font Information:

GetFontInfo, GetFontGlobals, GetFGSize

Three calls are included for gathering information on the current font.

GetFontInfo returns information in the following record:

```
FontInfoRecord =  
START  
ascent:    INTEGER;  
descent:   INTEGER;  
widMax:    INTEGER;  
leading:   INTEGER;  
End;
```

These values have been modified, if necessary, to reflect style modifications currently in effect (at this time that simply means that if bolding is on, widMax has been increased by 1).

GetFontGlobals returns a variable-length record:

```
FontGlobalsRecord =  
START  
family:    INTEGER;
```

## Fonts And Text In QuickDraw II

**style:** INTEGER;  
**size:** INTEGER;  
**version:** INTEGER;  
**widMax:** INTEGER;  
**fbrExtent:** INTEGER;

{additional fields may be present here}

END;

**widMax** is taken from the embedded Mac font; all the others are from the Cortland header. They are taken directly from the font, and not modified, regardless of any style modifications in effect.

We expect in the future to add more information to the FontGlobalsRecord. In order to warn the application, we have the call GetFGSize. It returns the length in bytes of the FontGlobalsRecord - currently 12 (as of QD II V.01.01). Future versions of QD II may add extra information at the ends of their FontGlobalsRecords, but they will maintain the documented fields and ordering of earlier versions for compatibility.

## Appendix E Change History

- First Publication**                      **June 15, 1985**
- Second Publication**                      **July 10, 1985**  
Added summary of hardware.  
Added Global Environment calls.  
Fixed typos reported by Harvey.  
A few GrafPort calls were made more like QuickDraw. QuickDraw names were adopted wherever there were questions.
- Third Publication**                      **July 25, 1985**  
Added commentary about initial review.  
Added Line Drawing calls.  
Raised questions of code size.
- Fourth Publication**                      **September 17, 1985**  
Mode is part of GrafPort  
Patterns are private.  
The bounds rect is private  
  
PenPat => SetPenPat  
BackPat => SetBackPat  
GetPenPat  
GetBackPat  
Appendix C
- Fifth Publication**                      **September 25, 1985**  
General Clarifications  
Clipping
- Sixth Publication**                      **October 29, 1985**  
General Clarifications  
GrafPort is no longer private
- Seventh Publication**                      **November 21, 1985**  
Adjusted to new tool locator specifications  
Adjusted to new memory manager specifications  
Added section on calling conventions  
Added utility calls to ERS  
Removed low level slab and slice routines and low level clipping routine information from this document.  
SetPenPat => PenPat  
SetBackPat => BackPat
- Eighth Publication**                      **December 3, 1985**  
Refined Color Table  
Removed Scroll Rect from Core Routines (they will be part of the expanded routines now).  
Updated Appendix C

Ninth Publication                      January 15, 1986

Changed the Name to QuickDraw II  
Removed Text Calls from this document.  
Added Region calls.  
Added ScrollRect  
Updated Appendix C

Tenth Publication                      March 5, 1986

Made GrafPort private and added calls for accessing fields.  
Added PenMask to grafport and drawing definition.  
Added calls for PenMask.  
Redefined codes for transfer modes. Added two text transfer modes.  
Added information on the Cursor data structure.  
Added information on Customizing QuickDraw operations.  
Added Text Calls and appendix on font definition. Added new font field to GrafPort.  
Added UserField and System Field to GrafPort.  
Added GetAddress Call.  
Updated Appendix C.

Eleventh Publication                      April 4, 1986

Added Polygon calls.  
Added SetIntUse call.  
Added Get & Set SysFont calls.  
Added Get & Set VisRgn calls.

Twelfth Publication                      April 25, 1986

Reorganized all the calls.  
Augmented introductory information  
Added missing calls:  
    ClearScreen  
    GrafOn  
    GrafOff  
    GetClipHandle  
    SetClipHandle  
    GetVisHandle  
    SetVisHandle  
    InitCursor  
Changed the way inputs are described.  
Changed input to SetRandSeed from integer to long.

Thirteenth Publication                      July 15, 1986

Added information on calls in RAM.  
Added information on Fonts.

Fourteenth Publication                      August 11, 1986

Corrected many errors reported by testing.  
Removed mention of the NotActive error. QuickDraw no longer supports this.  
Calling QuickDraw when it is not active leads to unpredictable results often fatal to the system.