



PDF417 Encoder

Version 2.2.1

Programmer's Manual

Silver Bay Software LLC
100 Adams Street
Dunstable, MA 01827
Phone: (800) 364-2889
Fax: (888) 315-9608
support@silverbaysoftware.com

Document Version 20091018

The information in this manual is subject to change without notice and should not be construed as a commitment by Silver Bay Software LLC. Silver Bay Software assumes no responsibility for any errors that might appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Products or brand names used herein are trademarks or registered trademarks of their respective companies

Copyright © 2009, Silver Bay Software LLC.
All rights reserved.

Table of Contents

1	INTRODUCTION.....	1
1.1	CONTENTS OF THIS MANUAL.....	1
1.2	BASIC FEATURES OF THE ENCODER.....	1
2	PDF417 SYMBOLOGY OVERVIEW	3
2.1	ANATOMY OF A PDF417 SYMBOL.....	3
2.1.1	<i>Symbol Rows</i>	3
2.1.2	<i>Modules</i>	4
2.1.3	<i>Codewords</i>	4
2.1.4	<i>Start Pattern, Stop Pattern and Row Indicators</i>	6
2.2	PDF417 SYMBOL SIZE.....	7
2.2.1	<i>Calculating the Size of a Particular Symbol</i>	7
2.2.2	<i>Determining Symbol Size from a Given Area</i>	8
2.2.3	<i>Module Size</i>	8
2.3	ERROR CORRECTION.....	9
2.4	TRUNCATED OR COMPACT PDF417.....	10
2.5	CHARACTER SET ISSUES.....	10
3	CONTROLLING THE GENERATION OF A PDF417 SYMBOL.....	11
3.1	CONTROLLING THE INTERNAL ENCODING METHOD.....	11
3.1.1	<i>Binary Mode</i>	11
3.1.2	<i>Optimized Mode</i>	11
3.2	CONTROLLING PDF SYMBOL SIZE.....	12
3.2.1	<i>Module Size - Font Rendering</i>	12
3.2.2	<i>Module Size - Graphics Rendering</i>	12
3.2.3	<i>Quiet Zones</i>	13
3.2.4	<i>Row and Column Control</i>	14
3.2.5	<i>Aspect Ratio Control</i>	14
3.2.6	<i>Error Correction Parameters</i>	15
3.3	INPUT DATA FORMAT.....	16
3.4	DEFAULT PARAMETERS.....	16
4	USING THE PDF417 ENCODER WITH THE C LANGUAGE.....	18
4.1	ENCODER OPERATION.....	18
4.1.1	<i>Encoder Input and Output</i>	18
4.1.2	<i>Font Rendering</i>	18
4.1.3	<i>Overview of the Encode Process</i>	19
4.2	ALLOCATING WORKING MEMORY.....	20
4.3	INITIALIZE WORKING MEMORY.....	20
4.4	SET ENCODER PARAMETERS.....	21
4.5	ENCODING DATA.....	21
4.6	ERROR CORRECTION AND RETRIEVAL OF SYMBOL DATA.....	22
4.7	RENDERING THE SYMBOL.....	22
4.8	PRINTING THE SYMBOL.....	23
4.9	RESULT CODES.....	23
5	C LANGUAGE API FUNCTIONS	24
5.1	INITIALIZATION PROCESS.....	24
5.1.1	<i>PdfInit</i>	24
5.1.2	<i>PdfClear:</i>	26
5.1.3	<i>PdfGenParamSet:</i>	27
5.1.4	<i>PdfGenParamGet:</i>	30
5.1.5	<i>PdfAspectRatioSet:</i>	31

5.1.6	<i>PdfAspectRatioGet:</i>	33
5.2	ENCODING PROCESS	33
5.2.1	<i>PdfEncodeOptimal</i>	33
5.2.2	<i>PdfEncodeBinary:</i>	35
5.2.3	<i>PdfRetrieveSymbol:</i>	36
5.3	RENDERING PROCESS	37
5.4	RENDERING CALLBACK ROUTINES.....	38
5.4.1	<i>Output Callback Functions</i>	38
5.4.2	<i>PdfSinkMemory</i>	39
5.4.3	<i>PdfSinkStream</i>	40
5.4.4	<i>PdfSinkFd</i>	41
5.5	DEVICE-INDEPENDENT BITMAP RENDERING ROUTINES	42
5.5.1	<i>PdfDIBQuery</i>	42
5.5.2	<i>PdfDIBRender:</i>	44
5.6	TAGGED IMAGE FILE FORMAT (TIFF) RENDERING ROUTINES	46
5.6.1	<i>PdfTIFFQuery:</i>	46
5.6.2	<i>PdfTIFFRender:</i>	47
5.7	FONT BASED RENDERING ROUTINES	49
5.7.1	<i>PdfFontInitRender</i>	49
5.7.2	<i>PdfFontRender:</i>	50
5.8	EBCDIC-TO-ASCII AND UTILITY ROUTINES.....	53
5.8.1	<i>PdfEtoA</i>	53
5.8.2	<i>PdfSet</i>	54
6	COBOL LANGUAGE API	55
6.1.1	<i>Overview of the Encode Process</i>	55
6.1.2	<i>Font Information Initialization</i>	55
6.1.3	<i>Encoder Parameters Initialization</i>	56
6.1.4	<i>COBOL Output record Initialization</i>	58
6.1.5	<i>Encoding Data</i>	60
6.1.6	<i>Printing the Symbol</i>	61
6.1.7	<i>Result Codes</i>	61
7	COBOL LANGUAGE API FUNCTIONS	62
7.1.1	<i>PDFINITF</i>	62
7.1.2	<i>PDFENCOD</i>	65
8	USING THE PDF417 ENCODER WITH OTHER LANGUAGES	69
8.1	RPG	69
8.2	OTHER LANGUAGES	69
9	FONT AND PRINTING-RELATED INFORMATION	70
9.1	PDF417 FONT BASICS	70
9.2	THE CHARACTER SET	70
9.3	MODULE SIZE	72
9.4	FONT METRICS	72
9.4.1	<i>AFP Fonts</i>	73
9.4.2	<i>HP PCL Fonts</i>	73
9.4.3	<i>Xerox Fonts</i>	74
9.5	IBM ADVANCED FUNCTIONAL PRINTING (AFP).....	76
9.6	XEROX METACODE/JSL	76
9.7	HEWLETT-PACKARD PRINTER CONTROL LANGUAGE (HP-PCL).....	76
9.8	AS/400 DDS	78
10	APPENDIX	79
10.1	API RETURN VALUES.....	79

10.2	SYMBOLGY TECHNICAL SUMMARY	81
10.3	FONT INITIALIZATION VALUES.....	82



PDF417 Encoder

Version 2.2.1

Programmer's Manual

1 Introduction

1.1 Contents of this Manual

This manual is broken into four sections:

- an introduction, ,
- an overview of the PDF417 symbology,
- a programmer's reference describing the API's, and
- a printing guide.

The introduction provides a quick overview of the PDF417 symbol and a general discussion on how to program with the Silver Bay Software LLC PDF417 encoder. The symbology overview describes how a PDF417 barcode is constructed, and provides information on how to get the barcode sized appropriately. The programmer's reference section provides the specific details of the API's for each of the supported programming languages. Finally, the printing section provides guidelines for formatting the output of the encoder in a variety of print environments, including AFP, Metacode, and HP-PCL.

1.2 Basic Features of the Encoder

The Silver Bay Software PDF417 encoder is designed to make it easy and straightforward to convert binary or ASCII data into a printable PDF417 symbol. The encoder offers the following features:

- Control of the minimum and maximum physical dimensions of the symbol to be generated.
- Control over the amount of error correction applied to the symbol.
- Choice between two encoding modes:
 - A binary mode, which is faster, but will tend to produce a slightly larger barcode.
 - An "optimized" mode, which will produce the smallest-possible symbol, at the cost of some additional processing time.

In addition, to be compatible with as many different computer and printing environments as possible, the encoder offers two different means via which the actual barcode is returned to the application programmer:

- All the API's offer "font rendering." In this mode, the output of the encoder is a rectangular array of characters which, when printed with one of the supplied custom fonts, will result in the image of the PDF417 symbol.
- The C language API also offers direct graphics rendering, in which a bitmap of the PDF417 symbol can be returned in one of the following bitmap formats:
 - Microsoft Windows Bitmap (BMP) format
 - Tagged Image File Format (TIFF)

If font rendering is being used, the basic steps for using the encoder are as follows:

1. Based on the type of printer being used, and the method via which it is connected to the computer, make the custom PDF417 font available to the printer.
2. If necessary, call the encoder initialization function with the correct character values. The values you use are based on your specific printing configuration. In most cases, this step is optional.
3. For each PDF417 symbol to be printed:
 - a. Place the data to be encoded into one of the structure or record formats supported by the API functions.
 - b. Call the appropriate encoder API.
 - c. Check the return code to ensure that the encode operation succeeded.
 - d. Send the appropriate command to invoke the PDF417 font on your printer.
 - e. Send the characters returned by the encoder to the printer.
 - f. Return the printer to the "normal" font.

For graphics rendering, the process is similar:

1. For each PDF417 symbol to be printed:
 - a. Place the data to be encoded into one of the structure or record formats supported by the API functions.
 - b. Call the appropriate encoder API.
 - c. Check the return code to ensure that the encode operation succeeded.
 - d. Call the appropriate rendering function to obtain the output graphic.
 - e. Send the graphic to the printer.

A number of sample programs have been provided with the distribution media as well, demonstrating the use of the PDF417 encoder.

2 PDF417 Symbology Overview

Portable Data File 417, or PDF417, is a multi-row, variable-length symbology offering high data capacity and error-correction capability for real-world applications where portions of symbols can get destroyed in handling. PDF417 symbols can be scanned by linear scanners, raster laser scanners, or two-dimensional imaging devices. In 1994, AIM USA, an affiliate of AIM International, standardized PDF417 in its "Uniform Symbology Specification – PDF417".



Figure 1 - Sample PDF417 Symbol

2.1 Anatomy of a PDF417 Symbol

In order to fully understand the process of controlling the size of PDF417 symbols, it is necessary to delve into the internal terminology used to describe the symbology itself, and the physical anatomy of the symbol.

2.1.1 *Symbol Rows*

Inspected closely, a PDF417 symbol looks like a set of stacked one-dimensional bar codes. Each of these is referred to as a *row*. Each individual row is relatively small, and consists of individual bars and spaces. Figure 2 shows the first three rows of the symbol from Figure 1, separated so that they can be more easily seen.



Figure 2 - Individual Rows of a PDF417 Symbol

If we “zoom in” on the barcode, as shown in Figure 3, we can easily see the individual rows, and the bars and spaces that make them up.

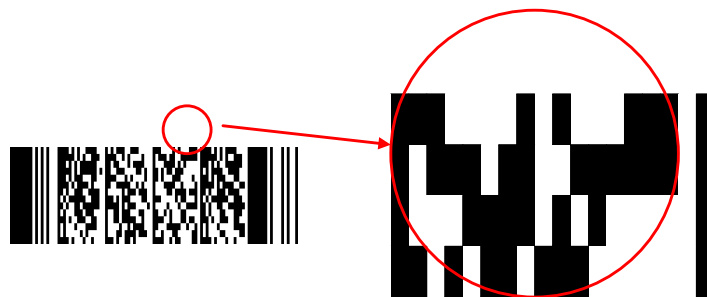


Figure 3 - Magnified Portion of PDF417 Symbol

2.1.2 Modules

In the first and third rows Figure 3, it is fairly easy to see the smallest black and white rectangles that are used in making up this particular barcode. It may not be quite as obvious that all of the bars and spaces, including the wider ones, are actually composed of multiple rectangles, all of which are the same size. Figure 4 shows the magnified portion of the barcode again, but this time with the individual rectangles outlined. As you can see, what appear to be wide or narrow white or black areas in the barcode are actually composed of multiple, adjacent rectangles of the same color.

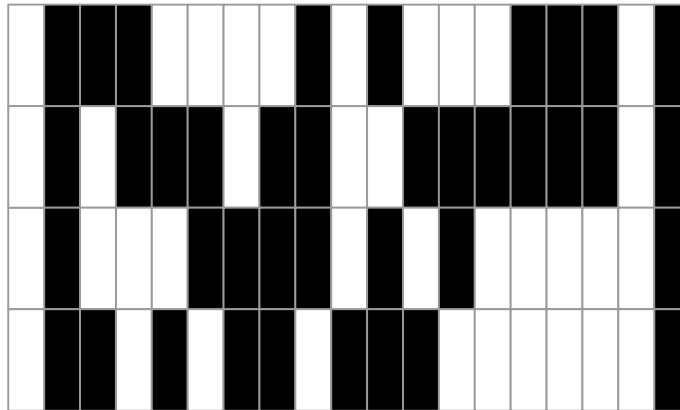


Figure 4 - Individual Modules in a PDF417 Symbol

These individual rectangles are referred to as *modules*. They represent the smallest unit of print in a PDF417 symbol. Each individual row in the PDF417 symbol is the height of a single module, so the height of the overall symbol is the number of rows times the module height.

2.1.3 Codewords

Within the PDF417 symbol, modules are organized in groups of 17. Each group of 17 modules is referred to as a *codeword*. Codewords always start with at least one black module, and always end with at least one white module, and there are always exactly four “bars” and four “spaces” (each of which is made up of one or more modules) in the codeword¹.

Figure 5 shows the same region of the PDF417 symbol as Figure 4, with the 17 modules that make up one of the codewords in the first row outlined. The white module to the left of this codeword is the last module of the previous codeword, while the black module to the right is the first module of the next codeword. The codeword structures on the subsequent rows of the symbol are easy to pick out – notice again that each contains 17 modules, organized into four bars and four spaces.

¹ The fact that there are four bars and four spaces making up seventeen modules is the origin of the “417” of PDF417. The “PDF” stands for “Portable Data File.”

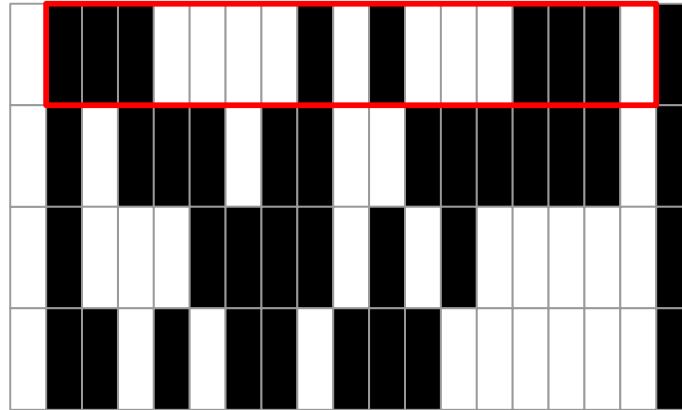


Figure 5 - A Codeword in the PDF417 Symbol

Codewords are significant because the internal encoding rules for the barcode translate the text or binary data being placed into the PDF417 symbol into a series of codewords. The actual process via which this is done is complex, but the programmer does not need to worry about this – the encoder software handles all these details. All that needs to be understood is that text and data get converted into codewords, and then the codewords are placed into the physical barcode.

There are actually two types of codewords in a PDF417 barcode. The codewords mentioned previously are referred to as *data codewords*, since they encode the data being transmitted by the PDF417 symbol. The second type of codewords are called *error correction codewords*, or *ECC codewords*.

When a PDF417 barcode is printed and later scanned, there is no guarantee that every single module will be recovered correctly by the scanner. Barcodes can, of course, be damaged, plus scanners may not perfectly scan the barcode². In order to prevent this from resulting in an unrecoverable symbol, PDF417 uses a technique called Reed-Solomon Error Correction. Essentially, this involves adding extra, specially calculated information to the barcode. When the barcode is scanned, this information can be examined mathematically, and can be used to fill in missing information or repair information that was damaged.

The codewords in a PDF417 symbol are arranged in a rectangle in the center of the symbol sometimes referred to as the *data region*. Figure 6 shows the location of the codewords within our sample PDF417 symbol. As you can see, in this particular symbol, they are arranged in two columns. The symbol itself has 14 rows, so there are a total of 28 codewords in this symbol. PDF417 symbols can have between 1 and 30 columns of codewords, and between 3 and 90 rows, allowing considerable variation in both the amount of data the symbol can hold and also its shape. Note that not every combination of row and column is allowed, however, since an individual PDF417 symbol is restricted to a maximum of 928 total codewords.

² PDF417 was specifically designed to be scanned by laser scanners, but there is no guarantee, for example, that the laser beam will cross each and every module in the barcode as it scans back and forth.

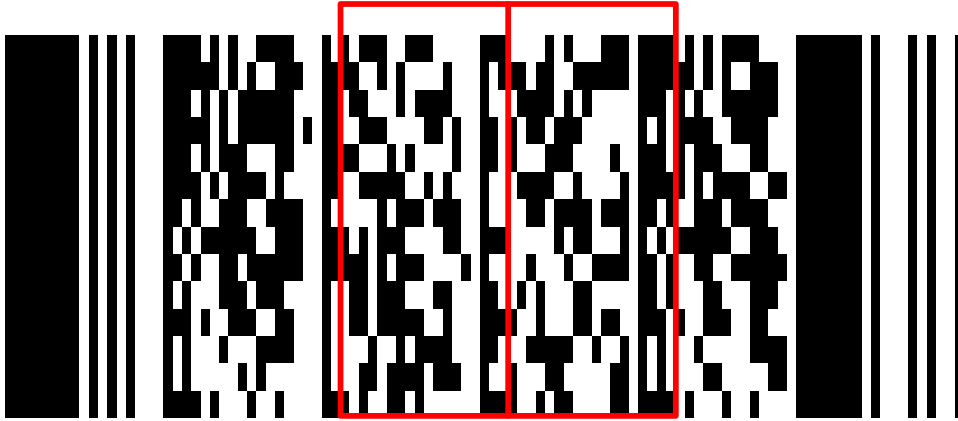


Figure 6 - Codewords in the Data Region of the PDF417 Symbol

2.1.4 Start Pattern, Stop Pattern and Row Indicators

Figure 7 shows the final portions of a PDF417 symbol. Because these areas do not carry data, they are sometimes referred to as “overhead” within the symbol.

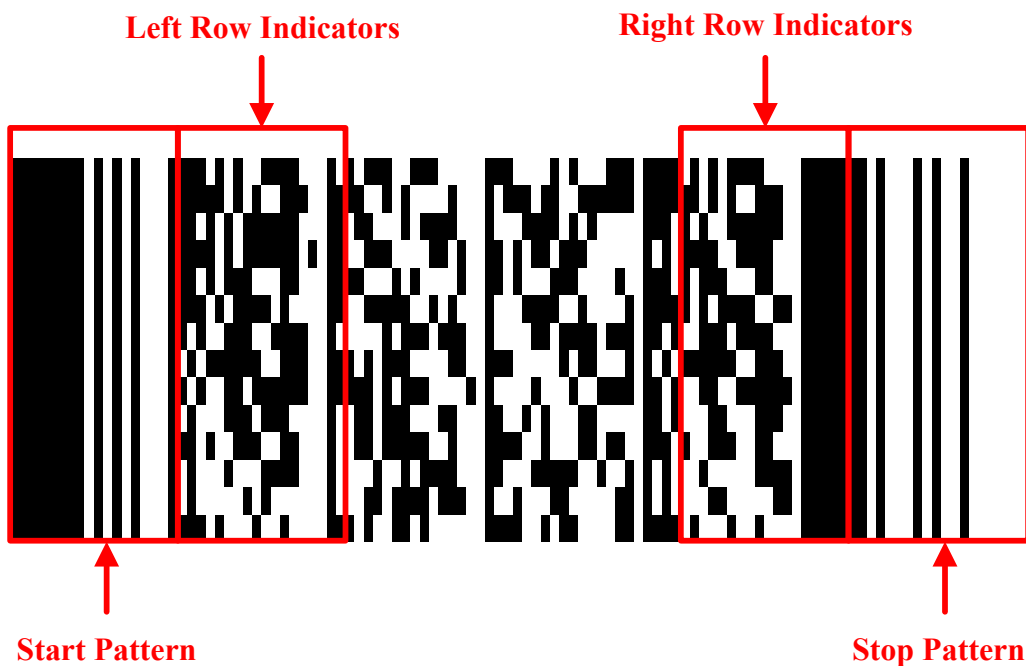


Figure 7 - PDF417 Symbol "Overhead" Areas

Every row in a PDF417 symbol begins with a special bar-space pattern called a *start pattern* and ends with another special bar-space pattern called a *stop pattern*. The patterns are identical on all rows, so they appear as solid vertical areas. These areas help a laser scanner detect when the scanning beam crosses into and out of a symbol, and help an imaging scanner find the symbol.

Just inside the start and stop patterns are the *left row indicators* and *right row indicators*. These have the same general appearance as codewords, but are constructed differently and do not carry data. Instead, these provide the following information:

- Which individual row within the barcode this is
- The number of rows in the barcode
- The number of columns in the data region of the symbol
- The error correction level of the symbol

Each individual row encodes its own row number, while the remainder of the information is encoded multiple times, spread across the various rows. The scanner uses this information both to recover the general shape of the symbol, but also to align itself to the rows of a symbol. A laser scanner beam, in particular, may cross the symbol diagonally. By knowing on what row the beam entered the symbol and on what row it exited, the scanner can usually compute the path the beam took through the symbol.

2.2 PDF417 Symbol Size

2.2.1 *Calculating the Size of a Particular Symbol*

The size of a PDF417 symbol can be determined given the following four quantities:

- The number of data columns in the symbol (between 1 and 30)
- The number of rows in the symbol (between 3 and 90)
- The height of a module
- The width of a module

Each data column is made up of codewords, each of which is 17 modules in width. The start pattern and row indicators are also 17 modules in width, while the stop pattern is 18 modules in width. Thus, there are a total of 69 modules worth of overhead (start, left row indicator, right row indicator and stop pattern) on each row in addition to the data codewords.

When determining the height and width of a symbol, consideration must also be given to the required *quiet zone*. The PDF417 standard requires an area of white space to the left and right of the symbol so that the scanner can accurately identify the start and stop patterns. (Obviously, if background graphics merged into either of these patterns, it would make decoding difficult or impossible.) The standard requires at least two modules worth of white space on each side of the symbol.

Thus, the width of the printed portion of a symbol will be:

$$\text{printed_symbol_width} = (\text{num_data_columns} * 17 + 69) * \text{width_of_module}$$

while the overall width that must be allotted for a symbol must be at least:

$$\text{total_symbol_width} = (\text{num_data_columns} * 17 + 73) * \text{width_of_module}$$

since four additional module widths must be allotted for the quiet zone.

Since each row is equal in height to the module height, the height of the symbol is easier to calculate:

```
symbol_height = num_rows * height_of_module
```

Thus, for example, our sample symbol, which has 14 rows and 2 data columns, if printed using modules that were 0.010 inches wide and 0.030 inches high, would be:

```
printed_symbol_width = (2 * 17 + 69) * 0.010 = 1.03 inches
total_symbol_width = (2 * 17 + 73) * 0.010 = 1.07 inches
symbol_height = 14 * 0.030 = 0.42 inches
```

2.2.2 Determining Symbol Size from a Given Area

The formulas above provide the size of a symbol if you have already determined the number of data rows and data columns. The encoder, however, requires you to indicate sizes to it in terms of rows and columns, not in physical dimensions.

If you have a certain width available on your form and need to determine the number of columns that will fit in it, you can use the following formula:

```
modules_available = INTEGER(width_available / width_of_module)
data_columns = INTEGER((modules_available - 73) / 17)
```

where the INTEGER() function means “largest integer less than” the enclosed quantity, and the width available includes the space required for the quiet zone.

The calculation above assumes that we are going to include the quiet zone in the generated symbol. If, instead, we have enough white space on the printed paper, and thus are not going to have the encoder reserve space in its output, then the calculation becomes:

```
modules_available = INTEGER(width_available / width_of_module)
data_columns = INTEGER((modules_available - 69) / 17)
```

Similarly,

```
rows_available = INTEGER(height_available / height_of_module)
```

As mentioned earlier, the maximum number of data columns that can be put into in a PDF417 symbol is 30, and the maximum number of rows is 90.

2.2.3 Module Size

Choosing the correct module size is one of the most critical items in designing a system that includes PDF417 symbols.

The smaller the module size, the smaller the barcode, and the more data can be encoded in a specific area. The larger the module size, however, the easier the barcode is to scan. In addition, larger module sizes are less sensitive to problems caused by subtle printing issues, like the tendency of ink to spread somewhat with inkjet printers.

Further, “tall” modules make it easier for people to align a laser scanner to read a symbol, since it gives more “diagonal” distance. “Short” modules, on the other hand, allow more rows to be placed in a given area.

The AIM specification for PDF417 recommends that modules be at least 0.010 inches wide and 0.030 inches high, and that modules be at least 3 times as high as they are wide.

In choosing a module size, however, consideration must also be given to the resolution of the printer being used to print the barcode – modules should always be designed to be an integer number of printer pixels. If this is not done, the module sizes will not be uniform across the symbol.

Thus, the following module sizes are common:

Printer Resolution	Module Size (pixels)		Module Size (inches)		Module Size (mm)	
	Width	Height	Width	Height	Width	Height
240 dpi	2	6	0.0083"	0.0250"	0.210 mm	0.635 mm
240 dpi	3	9	0.0125"	0.0375"	0.317 mm	0.952 mm
300 dpi	3	9	0.0100"	0.0300"	0.254 mm	0.762 mm
300 dpi	4	12	0.0133"	0.0400"	0.338 mm	1.016 mm

Note that, when using the Silver Bay Software encoder's font rendering option, the particular font chosen determines the module size. If using the graphics rendering option, the size of the module in pixels, and the resolution specified for the graphics image, determines the module size.

2.3 Error Correction

As mentioned in Section 2.1, a PDF417 symbol contains error correction codewords in addition to data codewords. The number of error correction codewords in a particular symbol cannot be chosen arbitrarily. Instead, one of a predetermined number must be used.

Internally, each symbol uses one of 9 "levels" of error correction. Increasing the error correction by one level doubles the number of error correction codewords that are included in the symbol. This, in turn, roughly doubles the amount of damage that the symbol can tolerate. Thus, from a symbol robustness point of view, using a higher level of error correction is advantageous. Doing this, however, increases the size of the symbol and, at the highest levels, limits the total amount of data that can be encoded in the symbol, since the total number of codewords (data plus error correction) that can appear in a symbol is limited.

The specific error correction (ECC) levels available are:

ECC Level	Number of ECC Codewords
0	2
1	4
2	8
3	16
4	32
5	64
6	128
7	256
8	512

The encoder allows the programmer to manually specify the error correction level to be used. Alternately, the encoder can choose the level automatically based on the amount of data in the symbol.

The PDF417 specification recommends the following minimum levels of error correction for a symbol:

Number of Data Codewords	ECC Level	Total Resulting Codewords
1 – 40	2	10 – 49
41 – 160	3	58 – 177
161 – 320	4	194 – 353
321 – 863	5	386 – 928

The observant reader will note that there is one extra codeword in the numbers in the right-hand column. Each PDF417 symbol has one extra codeword in it in addition to the data and ECC codewords. This code is used for internal purposes.

2.4 Truncated or Compact PDF417

For small PDF417 symbols, the start and stop patterns and the row indicators represent a significant amount of overhead in terms of print area. A variant of PDF417, referred to as “Truncated PDF417” in the AIM specification and “Compact PDF417” in newer specifications, exists which eliminates the right row indicators and all but one column module of the stop pattern. The encoder will generate symbols of either type, under the control of a parameter. The default is standard PDF417, which includes the right row indicators and stop pattern. This concludes the discussion of available user parameters.

2.5 Character Set Issues

The PDF417 symbology stores its internal information using the ASCII character set. While the ASCII character set is common on microcomputers and minicomputers, the EBCDIC character set is more common on mainframe systems.

On EBCDIC systems, textual data must be converted from EBCDIC to ASCII before it is actually placed into the barcode. The encoder library provides facilities to handle EBCDIC-to-ASCII conversion of the parameter data for the programmer, or the programmer may manually convert the data before presenting it to the encoder to process.

If the input data to the encoder is binary, or contains a mix of binary and text characters, then the application programmer must perform the EBCDIC-to-ASCII conversion prior to calling the encoder. If this is not done, the encoder may attempt to perform EBCDIC-to-ASCII conversions on the binary data, thus corrupting the barcode.

3 Controlling the Generation of a PDF417 Symbol

Unlike some other symbologies, PDF417 symbols can be designed in a variety of heights, widths and resolutions. It also supports a wide variety of data formats. The PDF417 encoder provides the programmer with a great deal of control over the symbol that is generated. This is done by setting various control parameters. When you initialize the PDF417 encoder, all parameters are assigned a default value. Thus, if you do not need fine control over the encoder's output, you may simply use it without setting any explicit parameter values. This section of the document describes each of the programmable parameters, its effect on the generated symbol, and why you might want to use a value other than the default.

3.1 Controlling the Internal Encoding Method

As was mentioned earlier, the process of converting text or binary data into the codewords that make up the PDF417 symbol can be a complex operation. In particular, for many data sets, there are multiple different methods that can be used to represent the data, due to the richness of the underlying PDF417 data encoding methods.

As a result, the Silver Bay Software PDF417 encoder provides the user with two basic approaches to encoding data – “binary mode” and “optimized mode.”

3.1.1 *Binary Mode*

Binary mode is the most straightforward encoding technique available in the PDF417 symbology. In binary mode, each group of six bytes is encoded into five codewords. If the data set to be encoded is not a multiple of six, the “extra” bytes are encoded one per codeword.

Binary mode has the following advantages:

- Any kind of data (text, binary, etc.) can be encoded.
- Because the calculations are very straightforward, it takes less processing than Optimized Mode to perform. Thus, the total amount of CPU time required to produce the symbol is smaller.
- The size of the symbol can be directly predicted based solely on the number of bytes of input data.

It also, however, has the following disadvantages:

- For data sets that are primarily alphanumeric, symbols encoded in Binary Mode are generally somewhat larger than those that would be created using the Optimized Mode.

3.1.2 *Optimized Mode*

In addition to its binary encoding mode, the PDF417 symbology has internal encoding methods that are designed to reduce the size of a symbol whose contents are text. Special modes exist to encode numeric sequences and text sequences, and the symbol can switch back and forth between these modes (and binary mode) at different points in the symbol in order to encode the data more compactly. As a result, there are at least two different ways to encode every alphabetic or punctuation character, and at least three ways to encode digits.

The Silver Bay Software encoder supports an “Optimized Mode” encoding selection. When this mode of encoding is used, the encoder will determine the best combination of encoding techniques in order to produce the smallest possible symbol.

Optimized Mode has the following advantages:

- Any kind of data (text, binary, etc.) can be encoded, since the optimizer will “fall back” to binary mode as required.
- The size of the symbol will be as compact as possible.

It also, however, has the following disadvantages:

- The calculations required are more complex than Binary Mode. As a result, the encoding process will be somewhat slower than for Binary Mode.

It is important to note that two data sets with the same number of characters but with different data may result in symbols that are of slightly different sizes. This can happen because, in one case, the various encoding modes available within the symbol may combine more efficiently than in another case, resulting in somewhat fewer codewords being generated. This is significant if you are trying to “tune” your symbol sizes to fit into a tight area. Just because one particular data set fits does not guarantee that another data set may not be just a little bigger. Therefore, you must always allow some margin for expansion, or you may find that the encoder will give you a “not enough space” error when encoding certain data sets³.

3.2 Controlling PDF Symbol Size

3.2.1 Module Size - Font Rendering

When using the font rendering process, the size of an individual module is controlled by the glyph size designed with the font.

3.2.2 Module Size - Graphics Rendering

When using the graphics rendering process, the portion of the PDF417 encoder that renders the symbol into a graphic format accepts four parameters to control the output file size:

- The module width in pixels.
- The module height in pixels.
- The amount to “shave” modules horizontally.
- The amount to “shave” modules vertically.

The module width is conventionally set at 10 mils, or 0.010 inches, and the module height at 30 mils, or 0.030 inches. The exact size that you choose is up to you; however there are two guidelines to remember:

³ The symbol generated in Optimized Mode will never be larger than the symbol generated in Binary Mode, since the optimizer will detect the cases where Binary Mode provides the most efficient encoding method.

- Under most circumstances, the module height should be at least three times the module width. If the module height is less than three times the module width the person scanning the symbol has to be much more careful to align the scanner with the symbol. Using a module height-to-width ratio of at least 3 allows a reasonable amount of tilt between the scanner and the symbol.
- The module height and width must be an integral number of pixels on your output device for maximum readability. Thus, if you are printing to a printer with a resolution of 360 dots per inch (DPI), you should probably pick a module width of 11.1 mils. (this is exactly 4 pixels). Trying to generate a PDF417 symbol with a 10 mil module width will cause some modules to end up 3 pixels and others 4, which disrupts the reading process to some extent. The encoder will only generate a symbol that uses an integral number of pixels for the module height and width; however you still must be careful that you do not introduce this type of distortion when you print the symbol. Thus, you will typically want to factor in the resolution of the printer as part of the symbol generation process.

During most printing processes, there is a tendency for the ink or toner to spread slightly. This means that when you print a PDF417 symbol, the black modules tend to end up slightly wider than expected, and the white modules slightly narrower. The encoder allows you to compensate for this ink spread by “shaving” the black modules. Choosing a non-zero value for the horizontal “shaving” amount, for example, causes the encoder to lay the symbol out using the specified module width, but then change part of each black module back to white.

The amount of shaving required depends entirely on the printing process and the type of stock being used, and can typically only be determined through experimental printing and measurement. With laser printers, this effect is minimal, and can usually be ignored. With inkjet printers, particularly on low-quality paper stock, the amount of ink spread can be significant. You may need to experiment to determine an appropriate setting.

You can determine the appropriate amount of horizontal shaving by printing a symbol with no shaving (zero) and with one pixel's worth of shaving. Look at the black and white vertical bars in the start patterns of both symbols, and choose the value for which the black and white bars are closest to the same width. In the vertical direction, again try with and without one pixel of shaving. If using one pixel's worth of shaving introduces a gap between successive rows of the symbol, go back to zero.

Only in the rarest of cases should you ever need more than one pixel's worth of shaving.

3.2.3 Quiet Zones

The PDF417 symbol requires a small amount of “white space” all the way around the symbol to ensure that the scanner can separate the symbol from its background. In barcode terminology, this is referred to as the “quiet zone.” The encoder provides you the option of including the quiet zone in the image by surrounding the actual barcode with white pixels. If you choose not to have the encoder include the quiet zone in the returned symbol, you must make sure that there is sufficient white space around the barcode when you print it.

3.2.4 Row and Column Control

Under normal circumstances, the PDF417 encoder will generate the smallest symbol (minimum number of rows and columns) that will contain the encoded data. This is done by setting the “use default” values. Some applications, however, may require that the generated symbol not exceed a certain width or a certain height. For example, you may be printing a PDF417 symbol onto a form that has a certain area reserved for the symbol. On this form, you may not care about the exact height of the symbol but may want the symbol to either have a specific width or a certain maximum width, a specific height or a certain maximum height.

In order to control the size and shape of the symbol, the PDF417 encoder accepts the following parameters:

- Minimum number of data columns to use. Set this if you want to ensure that the symbol is always a certain minimum width.
- Maximum number of data columns to use. Set this if you want to ensure that the symbol does not exceed a certain maximum width.
- Minimum number of rows to use. Set this if you want to ensure that the symbol is always a certain minimum height.
- Maximum number of rows to use. Set this if you want to ensure that the symbol does not exceed a certain maximum height.

You can force a specific width or height by setting the corresponding minimum and maximum to the same value. Remember that these parameters are in units of rows and columns. To translate this to actual printed dimensions, you need to use the formula given at the beginning of this section.

NOTE: Restricting the maximum height or width of a symbol may potentially reduce the amount of data that can be encoded into a symbol.

3.2.5 Aspect Ratio Control

Rather than controlling symbol size and shape via row and column counts, you may choose to control it via aspect ratio. The encoder provides the ability to provide “suggestions” to the encoder as to whether it generates a tall, narrow symbol, a relatively square symbol, or a short, wide symbol. This is controlled by a pair of values indicating the preferred aspect ratio. Thus, for example, if your form aesthetics suggest that the symbol should be approximately 3 times as wide as it is high, you may request a 3:1 aspect ratio symbol. When the data is encoded, the encoder will choose a row and column count that comes as close to this as possible.

Aspect ratio control and row/column control work together. The row and column minimum and maximum values are treated as absolutes. If the symbol cannot be fit into these constraints, an error is generated. Within the specified constraints, however, there is frequently more than one valid row/column choice. When this occurs, it is the aspect ratio information you specify that allows the encoder to choose a particular row/column combination as “better” than the others.

3.2.6 Error Correction Parameters

The PDF417 symbology allows the programmer a great deal of control over how much error correction is built into the symbol. While the default values are appropriate for many applications, if you anticipate that your printed symbol is likely to suffer damage, you may want to increase the amount of error correction that is built in. Conversely, if you know that the symbol will not be damaged, and will be read under relatively ideal conditions, you may wish to decrease the amount of error correction included. Be aware that the trade off is that a higher error correction level will result in a larger symbol.

The encoder accepts parameters to control the amount of error correction included in the symbol, the parameters work together to control what ECC level will be produced. The basic parameters are:

1. How the error correction is being specified. There are three choices:
 - a. Use the AIM recommended defaults. In this case, the encoder will automatically follow the recommendations in the PDF417 symbology standard.
 - b. Use a specific error correction level. In this case, the encoder will use the specific error correction level you specify in the value parameter.
 - c. By percent. If you select this option, you also specify a percent in the value parameter. The encoder will choose the lowest level of error correction that includes at least this percentage of ECC codewords. For example, if you specify 10%, and the symbol has 250 codewords, the encoder will include at least 25 error correction codewords. Since one of the levels above must be chosen, the encoder will, in this case, use level 4, which includes 32 error correction codewords.
2. Whether or not the encoder may “pad with ECC.”

Since the final PDF417 symbol must be rectangular, there sometimes arise situations in which the numbers of data and error correction codewords don't work out to an exact rectangle. In this case, the encoder automatically adds the required number of “pad” codewords to fill out the rectangle. In many cases, particularly in smaller symbols, the extra number of codewords that have to be added for padding turn out to be enough to increase the ECC level without increasing the size of the symbol. For example, suppose that you were generating a 10-column symbol, you had specified the use of ECC level 2, and had 64 data codewords. The 64 data codewords and 8 ECC codewords total to 72 codewords, which fill 7 full rows, plus two left-over codewords.

Because the data doesn't fit into 7 rows, the encoder will generate an 8-row symbol. This symbol will have a total of 80 codewords (8 rows times 10 columns), so the encoder will add 8 pad codewords to fill the $80-72=8$ unused codewords in the rectangle. In this case, however, instead of adding 8 pad codewords, the encoder could add 8 ECC codewords by switching from level 2 to level 3. The resulting symbol is the same size as one padded with “dummy codewords,” however it will tolerate more damage than if non-ECC codewords had been used to pad the symbol. Since it potentially provides a more robust symbol without increasing the symbol size, “pad with ECC” should, in general, be the programmer's choice, and thus is the default behavior of the encoder. There are, however, certain rare situations in which it is necessary to force a specific level of ECC, and not allow the encoder to change it. Because of this, the final decision as to whether the encoder is allowed to pad with ECC is left to you.

3.3 Input Data Format

In most applications, a PDF417 symbol will simply contain data. The AIM specification for PDF417, however, includes provision to embed “Global Label Identifier” (GLI) sequences mixed in with the data. These sequences are similar to escape sequences, in that they provide information *about* the data, rather than data itself. A newer, more portable representation for GLI sequences called “Extended Channel Interpretations” (ECI) or more generically “Extended Channel Escape” (ECE) has also been defined by the AIM standards body.

The PDF417 encoder supports data streams that have data only, streams with GLI sequences, and streams with ECE sequences. Because the three types of streams are formatted differently, the encoder needs to be told which type of stream it is receiving. Because GLI and ECE applications are rare at present, the default is a data-only stream.

3.4 Default Parameters

The following are the default parameter values that are set by the initialization process:

Parameter	Legal Values	Value Set by Default
Minimum Row Count	0, 3-90	0 = Don't care (any legal value)
Maximum Row Count	0, 3-90	0 = Don't care (any legal value)
Minimum Column Count	0, 1-30	0 = Don't care (any legal value)
Maximum Column Count	0, 1-30	0 = Don't care (any legal value)
Type of ECC Calculation	AIM Guidelines By Level By Percent	Use the AIM default guidelines
Pad with ECC	Allow Don't Allow	Allow
Data Stream Format	Data Only Data with embedded GLI sequences Data with embedded ECE sequences	Data only (no escape sequences)
Type of PDF417 Symbol	Standard Truncated/Compact	Standard (not Truncated/Compact)
Preferred Aspect Ratio		1:2 Twice as wide as high assuming modules are 3:1

If any of these defaults are not acceptable in your application, you may change them using the appropriate initialization or parameter API function.

Parameters may actually be set either before the data is encoded or after it is encoded. In most applications, the appropriate values are determined when the program is written, and thus are typically set before encoding. In some cases, however, you might want to encode your data first, and then set the parameters⁴. For example, you might have an application in which the PDF417 could go in one of two places on the form depending on its size. In one area, you can fit a small, rectangular symbol, and in another, a large wide one. In this case, you may want to encode the

⁴ This capability is only provided via the C language API.

data first, determine the size of the symbol, and then set the row/column restrictions or aspect ratio choice before you finish the encode process.

Determining the size of the symbol involves knowing the number of codewords required to encode the data and the error correction level. These two values will give the number of codewords that will be in the symbol. Once this information is known, you can choose the number of rows and columns required to render the symbol.

4 Using the PDF417 Encoder with the C Language

4.1 Encoder Operation

4.1.1 *Encoder Input and Output*

As you have no doubt surmised by now, the process of converting data to a two-dimensional barcode is called *encoding*. This is a sophisticated process that involves data validation, data compaction, and the insertion of error-correction information. When this document refers to the encoder, it is referring to the Silver Bay Software PDF417 encoder library. While the process of encoding the data is quite complex, Silver Bay Software has developed a set of simple to use functions for generating PDF417 symbols.

The input into the encoder functions is a work area which is allocated by the user, the number of bytes of data to encode, and a pointer to the data to encode. A more detailed discussion is provided in each of the language specific sections of this manual.

If the input data exceeds a single symbol there is an optional feature in the symbology called Macro PDF417 which provides a mechanism for the data to be split into blocks and be represented in more than one PDF417 symbol. A separate set of API functions handle Macro PDF417 symbols.

The PDF417 encoder is designed to be compatible with as many printing environments as possible. Since printing technologies, processes, and systems vary widely from computer system to computer system, the encoder does not directly print the symbol. Instead, the encoder operates in two phases. The first phase performs all the computations associated with the high-level encoding and error correction. The result of this phase is a PDFSYMBOL structure, which contains the numerical values of all the data and error correction codewords.

The second phase of the encode process is referred to as “rendering.” This process converts the intermediate PDFSYMBOL data to an appropriate format. The encoder supports a variety of output formats, including Microsoft Windows Device Independent Bitmaps (DIB or BMP format), Tagged Image File Format (TIFF) format, and font-based output. The encoder distribution kit contains API routines to automatically generate each of these formats from the PDFSYMBOL data.

Different environments may have different needs with respect to where the output data is placed. Some user environments will want the output to go to a file on disk, others to an area of memory. As a result, the data output process has been separated from the graphics data generation process, placing it under the control of the programmer. When the graphics rendering routines need to output data, they do so via a user-provided callback subroutine. The encoder distribution kit provides sample code for several different output callback functions. These samples handle the more common cases of disk and memory output.

4.1.2 *Font Rendering*

Since printing technologies, processes, and systems vary widely from computer system to computer system, the encoder also offers an option referred to as “font rendering.” Instead of a graphics file, the encoder returns a sequence of characters to the calling program. These

characters correspond to specially designed *code points* in a custom font provided with the encoder. When the returned characters are rendered (i.e., printed) in this font, the result is a PDF417 symbol. It is the responsibility of the application programmer to generate the appropriate print stream data to invoke the font on the printer and to send the characters returned by the encoder to the printer.

When using the font rendering API, the PDF417 encoder generates its output as a sequence of characters that must be matched to a corresponding font on the printer. Because of the varied ways in which a printer can be connected to a computer, and the corresponding varied character set transformations that can happen in this process, it is sometimes necessary to match the characters that the encoder outputs to the font characters actually installed on the printer. The encoder provides an initialization function that allows the actual characters output by the encoder to be properly set for the particular installation. Although this is rarely necessary, if you do not want to use the default character set, this initialization function should be called at the beginning of program execution. It does not need to be called for each symbol generated.

The output font characters will be placed into an output buffer defined by the user of the API. If this output area is defined too small, and the font characters to generate the symbol will not fit into the designated area, an error will be set and returned back through the API. To resolve this error, you must define the output area larger.

If you are going to use font rendering, be sure you read Section 9 for information related to how to print the symbol characters properly in your environment.

4.1.3 Overview of the Encode Process

There are essentially seven steps to using the encoder. Briefly, they are as follows:

1. Allocate working memory for the encoder.
2. Initialize the encoder working memory.
3. Set encoder parameters to control the nature of the symbol to be generated (optional).
4. Perform the data encode.
5. Apply the error correction codewords and retrieve the encoded data in a PDFSYMBOL structure.
6. Render the PDFSYMBOL data into one of the supported output formats.
7. Print the symbol. This step is the user's responsibility, and is not performed with functions provided by the encoder API.

If additional symbols need to be generated, the process may be repeated starting with either Step 3 (if the parameters need to be modified) or Step 4 (if the parameters do not need to be changed).

Each of these steps is described in more detail below. Manual pages for the individual API subroutines are provided at the end of this section.

4.2 Allocating Working Memory

The PDF417 encoder requires working memory in which to store temporary calculation results, partially encoded data, and the parameters provided by the user. In order to maximize the flexibility of the encoder, allocation of this working memory is the responsibility of the user.

The working memory is contained in a `PDFUSER` structure (defined in `pdfdefs.h`). An instance of this structure may be allocated in one of three ways:

- Globally, by declaring a `PDFUSER` structure as a global variable.
- Locally (on the stack), by declaring a `PDFUSER` variable inside a subroutine.
- Dynamically, by allocating a `PDFUSER` structure using “`malloc`”, “`new`”, etc.

Requiring the programmer to handle the working area allocation has the following advantages:

- The PDF417 encoder has no dependence on C run-time routines such as “`malloc`” and “`free`”.
- The programmer may choose the best way to allocate the memory for his environment.

The PDF417 encoder is thread-safe. Different threads of execution may be working on different symbols without collisions, provided that each thread uses a unique `PDFUSER` area.

Most of the routines in the API require that the address of the `PDFUSER` area be passed as the first parameter.

The optimizing version of the data encoder (`PdfEncodeOptimal`) requires more working memory than the “binary-only” version of the encoder (`PdfEncodeBinary`). This extra memory has been separated into a `PDFOPTWORK` structure so that users who do not require the optimizing encoder are not burdened with its extra memory requirements. Thus, if the optimizing encoder is used, an instance of the `PDFOPTWORK` structure must be allocated and its address passed to `PdfEncodeOptimal`. Unlike the `PDFUSER` structure, the `PDFOPTWORK` structure is only needed while `PdfEncodeOptimal` is running, and thus can be discarded after `PdfEncodeOptimal` returns, if necessary. No special initialization of this area is required.

4.3 Initialize Working Memory

The API includes a function `PdfInit` that will properly initialize the contents of a `PDFUSER` structure. `PdfInit` sets all the encoder parameters and aspect ratio values to appropriate default variables, and prepares the working memory area to receive encoded data.

IMPORTANT: You *MUST* call `PdfInit` to initialize the `PDFUSER` structure before calling any of the other API functions.

The API provides another function named `PdfClear` that is related to `PdfInit`. Situations may arise in which the encoding process fails to complete properly. One example would be if more data is passed to the encoder than will fit in a single symbol. Following a failure of this type, the `PDFUSER` structure may be left in an inconsistent internal state. To correct this, the user must call either `PdfInit` or `PdfClear`. The difference between the two is that `PdfClear` will clear out the results of the previous encode operation, but will not modify the encoder

parameters that may have been set. `PdfInit`, on the other hand, will reinitialize all the encoder parameters in the `PDFUSER` structure to their default values.

4.4 Set Encoder Parameters

Section 3.4 listed the default parameter values that are set by `PdfInit`.

If any of these defaults are not acceptable in your application, you may change them with a call to `PdfGenParamSet` or `PdfAspectRatioSet`. The current settings may be retrieved with `PdfGenParamGet` or `PdfAspectRatioGet` respectively.

4.5 Encoding Data

Prior to encoding, the input data stream might require conversion from EBCDIC to ASCII. The PDF417 symbology stores its internal information using the ASCII character set. This decision rests on the environment in which you are executing the encoder and what type of data the application is encoding. It is up to the user to make this decision and convert the input data into the required character set.

The actual data encoding process is performed using one of two functions. The two differ in their approach to the encoding process. The two functions are:

<code>PdfEncodeOptimal</code>	This function will generate the smallest possible symbol given the input data.
<code>PdfEncodeBinary</code>	This function will encode the data in the fastest possible manner, trading off speed for a somewhat larger symbol.

As described in Section 3.1, the PDF417 symbology provides multiple basic “modes” for encoding data. One mode is best for ASCII text, another for numeric data, and a third for binary data. Which mode is “best” for each character depends on the specific character sequences that precede and follow it.

The `PdfEncodeOptimal` routine uses a sophisticated dynamic programming algorithm to choose among the various ways of encoding the data. It is designed to choose the appropriate mode for each character so that the overall symbol has the absolute smallest number of data codewords. Thus, using this function will result in the smallest possible symbol. The tradeoff is that this function takes longer to run, and requires more working memory than `PdfEncodeBinary`.

The `PdfEncodeBinary` routine, on the other hand, emphasizes speed over symbol size. It always uses the binary compaction mode of PDF417, which can encode any 8-bit value, is very fast to compute, and uses the minimum amount of working memory. The binary compaction mode, however, achieves the poorest byte-to-codeword ratio of the available modes, so symbols encoded with this function tend to be larger than those encoded with `PdfEncodeOptimal`.

Which function you should use depends on your application requirements and the nature of your data. Unfortunately, it is difficult to determine, in advance, how much slower `PdfEncodeOptimal` runs and how much smaller the symbol will be, since this is entirely dependent on the nature of the data that is passed in. You may wish to experiment with both encoding options.

4.6 Error Correction and Retrieval of Symbol Data

Once your data is encoded and any parameters you want to change after the encode process are set, you call PdfRetrieveSymbol to apply the error correction desired and return the PDFSYMBOL data. This process has been separated from the data encode process to allow parameters to be changed after the number of data codewords has been determined, if desired. The PdfRetrieveSymbol function does the following:

- Determines the number of rows and columns in the final symbol, based on the sizing and aspect ratio parameters the user specifies.
- Determines the ECC level that will be used based on the ECC parameters the user specifies.
- Applies any pad codewords required to fill the symbol rectangle.
- Computes the actual ECC codewords and appends them to the end of the symbol data.
- Copies the final symbol data into a PDFSYMBOL structure that you provide.
- Clears the encoded data out of the PDFUSER area so that it is not necessary to call PdfClear or PdfInit again before encoding more data.

4.7 Rendering the Symbol

Given the symbol data returned by the PdfRetrieveSymbol function, you can now produce graphics data that you can use to print the symbol. Three output options are available in this version of the encoder:

- Microsoft Device Independent Bitmap (DIB / BMP) format (PdfDIBRender).
- Tagged Image File Format (TIFF) (PdfTIFFRender).
- Font-based output (PdfFontRender).

All three of the output options require that you provide a call-back function with a specific prototype. The prototype is defined in Section 5.4. The rendering function calls this function any time it needs to output data. This provides you with the maximum amount of flexibility as to how you want to store the data. To make things easier, the encoder includes source code for the most common types of callback functions, allowing output to files or to memory. You may either use this source as-is, or use it as a pattern from which to design your own callback functions.

The BMP and TIFF rendering routines require that you provide the module dimensions in pixels, and the amount of shaving to be performed on black modules. In addition, these routines require working memory in which to build scan lines. The amount of working memory required depends on the size of an individual line. The API includes functions (PdfDIBQuery and PdfTIFFQuery) that allow you to determine how much memory will be required, as well as the total number of bytes that will be output. The latter quantity is useful if you are outputting to memory, as it allows you to preallocate a block that will be big enough to hold the entire symbol. Refer to Section 4.1.3 and the description of the individual API for further detail on how the query function works together with the rendering function.

Font-based output produces a stream of characters that, when printed using a special font provided with the encoder, will produce a PDF417 symbol. Refer to the description of the individual API for more detailed information on how this is accomplished.

4.8 Printing the Symbol

Printing the symbol is your (the user's) responsibility. Because the printing requirements vary so much from system to system and application to application, the API provides no functions to perform the printing operation.

4.9 Result Codes

Each API function returns a status result value. If the encode and symbol generation is successful, the status should return `PERR_OK`. Refer to Section 10.1 for expected values and detailed definitions.

5 C Language API Functions

This section of the document provides detailed descriptions of each of the C Language API routines

5.1 Initialization Process

5.1.1 *PdfInit*

Description:

This function is used to initialize the encoder working area after it has been allocated. . It also sets all general user parameters and the aspect ratio parameters to their default values.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfInit( PDFUSERPTR pUserArea );
```

Arguments:

pUserArea a pointer to a PDFUSER structure.

A PDFUSER structure is defined as follows:

```
#define PDF_USER_AREA    3200            (defined in pdfdefs.h - number of unsigned longs)

struct sPdfUserArea
{
    unsigned long userArea[PDF_USER_AREA];
};
```

NOTES:

- The include file pdfenc.h contains all of the type definitions and prototypes for the encoder API functions (pdfenc.h includes pdfdefs.h).
- PdfInit must be called before calling any other encoder function, otherwise the encoder may not function properly.
- This function does not need to be called before every symbol encoded. It only needs to be called once as part of program initialization.
- This function makes a call to both PdfGenParamSet and PdfAspectRatioSet to set the default values.

Example:

```
void main(void)
{
    PDFUSER userArea;
    PDFERROR returnStat = PERR_OK;

    returnStat = PdfInit( &userArea );
    if( returnStat != PERR_OK )
    {
```

```
        printf("*** ERROR calling PdfInit() status = %d", returnStat);  
    }  
    . . . . .  
}
```

5.1.2 PdfClear:

Description:

This function is used to clear the PDFUSER structure without changing any of the general user or aspect ratio parameters. It should be used after an encoder function returns an error before attempting to encode another symbol.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfClear( PDFUSERPTR pUserArea );
```

Arguments:

pUserArea a pointer to a PDFUSER structure.

NOTES:

- The include file pdfenc.h contains all of the type definitions and prototypes for the encoder API functions (pdfenc.h includes pdfdefs.h).
- Refer to PdfInit for further documentation on structures.

Example:

```
void main(void)
{
    PDFUSER userArea;
    PDFERROR returnStat = PERR_OK;

    . . . . .

    returnStat = PdfClear( &userArea );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfClear() status = %d", returnStat);
    }

    . . . . .
}
```

5.1.3 PdfGenParamSet:

Description:

This function is used to set encoder general user definable parameters.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfGenParamSet( PDFUSERPTR pUserArea,
                                PDFPARAMCPTR pSetData );
```

Arguments:

pUserArea a pointer to a PDFUSER structure.
pSetData a pointer to a PDFPARAM structure.

All fields in the input structure PDFPARAM must contain either a valid value or be initialized to PDF_USE_DEFAULT, the encoder's default value.

A PDFPARAM structure is defined as follows:

```
struct sPdf417Parameters
{
    unsigned int minRows;            /* rows in symbol (3-90)            */
    unsigned int maxRows;            /*                                    */
    unsigned int minCols;            /* columns in symbol (1-30)        */
    unsigned int maxCols;            /*                                    */
    ECCTYPE eEccType;                /* Use default, level or percent    */
    unsigned int eccValue;            /* ecc level (0 - 8 ) or            */
    /*                                /* ecc percentage (1-100)            */
    PDFBOOL bNoPadExtraEcc;         /* If have room to increase ECC level, */
    /*                                /* what to do? False = increase if   */
    /*                                /* have room in symbol, True = don't */
    PDFDF    eDataFormat;            /* Flag to indicate data stream format. */
    /*                                /* escape sequence as GLI or ECI or   */
    /*                                /* no escape sequences in data       */
    PDFBOOL bIsTruncated;           /* 0 = Standard, 1 = Truncated symbol */
};
typedef struct sPdf417Parameters PDFPARAM;
typedef struct sPdf417Parameters PDFPTR *PDFPARAMPTR;
typedef const struct sPdf417Parameters PDFPTR *PDFPARAMCPTTR;
```

The individual structure members are defined as follows:

minRows	The minimum number of rows to use in the symbol.
maxRows	The maximum number of rows to use in the symbol.
minCols	The minimum number of columns to use in the symbol.
maxCols	The maximum number of columns to use in the symbol.
eEccType	An enumerated type, ECCTYPE (defined in pdfdefs.h). The table below lists the possible values:

Value	Meaning
ECC_DEFAULT	Use the AIM-recommended default error correction levels. NOTE: when this is selected, the <code>eccValue</code> parameter is ignored, and should be set to zero.
ECC_LEVEL	Use the specific ECC level specified in <code>eccValue</code> .
ECC_PERCENT	Use the ECC Percentage in <code>eccValue</code> .

NOTE: If `ECC_DEFAULT` is set, the `eccValue` parameter (below) must be set to 0 and will be ignored

<code>eccValue</code>	The meaning of this parameter depends on the value of <code>eEccType</code> . For <code>ECC_DEFAULT</code> , this parameter must be set to 0 and is ignored. For <code>ECC_LEVEL</code> , this parameter contains the ECC level (0-8). For <code>ECC_PERCENT</code> , it contains the ECC percentage (0-100).								
<code>bNoPadExtraEcc</code>	If this is set to <code>PFALSE</code> , the encoder is allowed to pad with ECC codewords. If it is set to <code>PTRUE</code> , the encoder is prohibited from doing this.								
<code>eDataFormat</code>	An enumerated type, <code>PDFDF</code> (defined in <code>pdfdefs.h</code>). The table below lists the possible values:								
	<table> <thead> <tr> <th>Value</th> <th>Meaning</th> </tr> </thead> <tbody> <tr> <td><code>PDF_DEFAULT_FORMAT</code></td> <td>The input is a simple data stream without any embedded escapes.</td> </tr> <tr> <td><code>PDF_GLI_FORMAT</code></td> <td>Output data in GLI format (<code>\nnn\nnn</code> or <code>\nnn\nnn\nnn</code>).</td> </tr> <tr> <td><code>PDF_ECI_FORMAT</code></td> <td>Output data in ECI format (<code>\nnnnnn</code>).</td> </tr> </tbody> </table>	Value	Meaning	<code>PDF_DEFAULT_FORMAT</code>	The input is a simple data stream without any embedded escapes.	<code>PDF_GLI_FORMAT</code>	Output data in GLI format (<code>\nnn\nnn</code> or <code>\nnn\nnn\nnn</code>).	<code>PDF_ECI_FORMAT</code>	Output data in ECI format (<code>\nnnnnn</code>).
Value	Meaning								
<code>PDF_DEFAULT_FORMAT</code>	The input is a simple data stream without any embedded escapes.								
<code>PDF_GLI_FORMAT</code>	Output data in GLI format (<code>\nnn\nnn</code> or <code>\nnn\nnn\nnn</code>).								
<code>PDF_ECI_FORMAT</code>	Output data in ECI format (<code>\nnnnnn</code>).								
<code>bIsTruncated</code>	If set to <code>PFALSE</code> , a standard symbol is generated. If set to <code>PTRUE</code> , a truncated symbol is generated.								

NOTES:

- The include file `pdfenc.h` contains all of the type definitions and prototypes for the encoder API functions (`pdfenc.h` includes `pdfdefs.h`).
- There is a Boolean typedef `PDFBOOL` for true and false values defined in `pdfdefs.h`.
- All values must be set to a valid value or `PDF_USE_DEFAULT` to use the default value.
- To force the column or row to be a specific value, set both the minimum and maximum values to be equal.
- The `PdfAspectRatioSet` API allows a user to set the aspect ratio parameters of a symbol.

Example:

```
void main(void)
{
    PDFUSER userArea;
    PDFPARAM userParam;
    PDFERROR returnStat = PERR_OK;

    . . . . .

    userParam.minRows = PDF_USE_DEFAULT;
    userParam.maxRows = PDF_USE_DEFAULT;
    userParam.minCols = 3;
    userParam.maxCols = 3;
    userParam.eccType = ECC_PERCENT;
    userParam.eccValue = 10;
    userParam.bNoPadExtraEcc = PFALSE;
    userParam.dataFormat = PDF_DEFAULT_FORMAT;
    userParam.bIsTruncated = PFALSE;

    . . . . .

    returnStat = PdfGenParamSet( &userArea, &userParam );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfGenParamSet() status = %d", returnStat);
    }

    . . . . .
}
```

5.1.4 PdfGenParamGet:

Description:

This function is used to retrieve the encoder's general user parameters.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfGenParamGet( PDFUSERPTR pUserArea,
                                PDFPARAMPTR pGetData );
```

Arguments:

pUserArea	a pointer to a PDFUSER structure.
pGetData	a pointer to a PDFPARAM structure.

NOTES:

- The include file `pdfenc.h` contains all of the type definitions and prototypes for the encoder API functions (`pdfenc.h` includes `pdfdefs.h`).
- The user must allocate a `PDFPARAM` structure to receive the values.
- Refer to `PdfGenParamSet` for further documentation on structures and enumerated values of the structure.
- There is another API called `PdfAspectRatioGet` which will retrieve the aspect ratio parameters.

Example:

```
void main(void)
{
    PDFUSER userArea;
    PDFPARAM recvr;
    PDFERROR returnStat = PERR_OK;

    . . . . .

    returnStat = PdfGenParamGet( &userArea, &recvr );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfGenParamGet() status = %d", returnStat);
    }

    . . . . .
}
```

5.1.5 PdfAspectRatioSet:

Description:

This function is used to set the encoder's preferred aspect ratio parameters.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfAspectRatioSet( PDFUSERPTR pUserArea,
                                   PDFARCPTR pAspectRatio);
```

Arguments:

`pUserArea` a pointer to a `PDFUSER` structure.
`pAspectRatio` a pointer to a `PDFAR` structure.

A `PDFAR` structure is defined as follows:

```
struct sPdfAspectRatio
{
    unsigned int moduleWidth;
    unsigned int moduleHeight;
    unsigned int symbolWidth;
    unsigned int symbolHeight;
};
```

The members of this structure are as follows:

<code>moduleWidth</code>	The width of an individual module.
<code>moduleHeight</code>	The height of an individual module.
<code>symbolWidth</code>	The desired width of the symbol..
<code>symbolHeight</code>	The desired height of the symbol..

Note that the specific values passed in are not important. The important information is the module height-to-width ratio and the symbol height-to-width ratio.

NOTES:

- The include file `pdfenc.h` contains all of the type definitions and prototypes for the encoder API functions (`pdfenc.h` includes `pdfdefs.h`).
- All values must be set to a valid value or `PDF_USE_DEFAULT` to use the default value.
- There is another API called `PdfGenParamSet` which allows for a user to set the general user parameters of a symbol.

Example:

```
void main(void)
{
    PDFUSER userArea;
    PDFAR aspectRatio;
    PDFERROR returnStat = PERR_OK;

    . . . . .

    aspectRatio.moduleWidth = 1;      /* modules 3 times as high as wide */
    aspectRatio.moduleHeight = 3;
    aspectRatio.symbolWidth = 2;     /* symbol 2 times as wide as high */
    aspectRatio.symbolHeight = 1;

    returnStat = PdfAspectRatioSet( &userArea, &aspectRatio );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfAspectRatioSet() status = %d",
               returnStat);
    }

    . . . . .
}
```

5.1.6 PdfAspectRatioGet:

Description:

This function is used to retrieve the encoder's aspect ratio parameters.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfAspectRatioGet( PDFUSERPTR pUserArea,
                                   PDFARPTR pAspectRatio );
```

Arguments:

pUserArea a pointer to a PDFUSER structure.
pAspectRatio a pointer to a PDFAR structure.

NOTES:

- The include file pdfenc.h contains all of the type definitions and prototypes for the encoder API functions (pdfenc.h includes pdfdefs.h).
- Refer to PdfAspectRatioSet for further documentation on structures.
- There is another API called PdfGenParamGet which will retrieve the general user parameters.

Example:

```
void main(void)
{
    PDFUSER userArea;
    PDFAR aspectRatio;
    PDFERROR returnStat = PERR_OK;

    . . . . .

    returnStat = PdfAspectRatioGet( &userArea, &aspectRatio );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfAspectRatioGet() status = %d",
              returnStat);
    }

    . . . . .
}
```

5.2 **Encoding Process**

5.2.1 PdfEncodeOptimal

Description:

This function will encode the data using the fewest number of codewords possible. Be aware that this method will result in the fewest number of codewords, but is usually slower and requires more work space memory. If size is the major concern, this method should be used.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfEncodeOptimal( PDFUSERPTR pUserArea,
                                   unsigned int nBytes,
                                   const void PDFPTR *pData,
                                   PDFOPTWORKPTR pOptWorkArea );
```

Arguments:

pUserArea a pointer to a PDFUSER structure.
nBytes number of bytes of data to encode.
pData a pointer to the data stream to encode.
pOptWorkArea a pointer to a PDFOPTWORK structure.

The PDFOPTWORK structure is defined as follows:

```
#define PDF_OPT_AREA    65200            (defined in pdfdefs.h - number of unsigned longs)

struct sPdfOptimalWorkArea
{
    unsigned long data[PDF_OPT_AREA];
};
```

NOTES:

- The include file pdfenc.h contains all of the type definitions and prototypes for the encoder API functions (pdfenc.h includes pdfdefs.h).
- pData is defined as a void pointer so that any type of data may be passed in.

Example:

```
void main(void)
{
    PDFUSER userArea;
    unsigned int nBytes;
    unsigned char inData[30];
    PDFOPTWORK optWorkArea;
    PDFERROR returnStat = PERR_OK;

    returnStat = PdfInit( &userArea );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfInit() status = %d", returnStat);
        exit(0);
    }

    strcpy(inData, "Silver Bay Software, LLC.");
    nBytes = strlen(inData);

    returnStat = PdfEncodeOptimal( &userArea, nBytes, inData, &optWorkArea );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfEncodeOptimal() status = %d",
              returnStat);
    }

    . . . . .
}
```

5.2.2 PdfEncodeBinary:

Description:

This function will encode the data using the binary compaction mode, which is the fastest technique. As a result, however, the symbol may be larger than if PdfEncodeOptimal were used. If speed is the major concern, then this method should be used.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfEncodeBinary( PDFUSERPTR pUserArea,
                                unsigned int nBytes,
                                const void PDFPTR *pData );
```

Arguments:

pUserArea	a pointer to a PDFUSER structure.
nBytes	number of bytes of data to encode.
pData	a pointer to the data stream to encode.

NOTES:

- The include file pdfenc.h contains all of the type definitions and prototypes for the encoder API functions (pdfenc.h includes pdfdefs.h).
- pData is defined as a constant void pointer so the user can pass in any data type pointer.

Example:

```
void main(void)
{
    PDFUSER userArea;
    unsigned int nBytes;
    unsigned char inData[30];
    PDFERROR returnStat = PERR_OK;

    returnStat = PdfInit( &userArea );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfInit() status = %d", returnStat);
        exit(0);
    }

    strcpy(inData, " Silver Bay Software, LLC.");
    nBytes = strlen(inData);
    returnStat = PdfEncodeBinary( &userArea, nBytes, inData );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfEncodeBinary() status = %d",
              returnStat);
    }

    . . . . .
}
```

5.2.3 PdfRetrieveSymbol:

Description:

This function chooses the actual height and width of the symbol, pads it if required, computes and appends the ECC codewords, and returns the resulting symbol data to the user. It then clears the PDFUSER area so that it is ready for the next encoding operation. If you are going to be using the font rendering routines, the row and column parameters will need to correspond to your receiving area for the font characters. This routine generates the symbol “look”, setting the rows and columns of the symbol based on the user parameters that are chosen. Refer to the font rendering API (`PdfFontRender`) for more information on how the symbol generated by this routine will need to reflect the output buffer requirements.

Prototype:

```
#include "pdfenc.h"

PDFERROR PDFAPI PdfRetrieveSymbol( PDFUSERPTR pUserArea,
                                   PDFSYMBOLPTR pSymbol );
```

Arguments:

```
pUserArea      a pointer to a PDFUSER structure.
pSymbol        a pointer to a PDFSYMBOL structure.
```

A PDFSYMBOL structure is defined as follows:

```
struct sPdfSymbol
{
    unsigned int  nRows;           /* rows in symbol (3-90)      */
    unsigned int  nCols;           /* columns in symbol (1-30)   */
    unsigned int  eccLevel;        /* ecc level (0-8)            */
    PDFBOOL       bIsTruncated;    /* Is Truncated symbol       */
    PDFCODEWORD   codeword[PDF_MAX_CODEWORDS];
};
typedef struct sPdfSymbol PDFSYMBOL;
typedef struct sPdfSymbol PDFPTR *PDFSYMBOLPTR;
typedef const struct sPdfSymbol PDFPTR *PDFSYMBOLCPTR;
```

The individual structure members are defined as follows:

```
nRows          The number of rows in the symbol.
nCols          The number of columns in the symbol.
eccLevel       The ECC level that was chosen for the symbol.
bIsTruncated   PTRUE if this is a truncated symbol, PFALSE for a standard symbol.
codeword       An array containing the codewords in the symbol.
```

NOTES:

- The include file `pdfenc.h` contains all of the type definitions and prototypes for the encoder API functions (`pdfenc.h` includes `pdfdefs.h`).
- Once the symbol has been retrieved, the PDFSYMBOL may then be used with any of the rendering functions to generate the desired output (see next section for details).

Example:

```

void main(void)
{
    PDFUSER userArea;
    unsigned int nBytes;
    unsigned char inData[30];
    PDFSYMBOL recvrSymbol;
    PDFERROR returnStat = PERR_OK;

    returnStat = PdfInit( &userArea );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfInit() status = %d", returnStat);
        exit(0);
    }

    strcpy(inData, "Silver Bay Software, LLC.");
    nBytes = strlen(inData);

    returnStat = PdfEncodeBinary( &userArea, nBytes, inData );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfEncodeBinary() status = %d",
            returnStat);
        exit(0);
    }

    returnStat = PdfRetrieveSymbol( &userArea, &recvrSymbol );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfRetrieveSymbol() status = %d",
            returnStat);
    }

    . . . . .
}

```

5.3 Rendering Process

The second phase of the encode process is referred to as “*rendering*”. This process converts the intermediate PDFSYMBOL data to an appropriate graphic format. There are many rendering formats for the PDF417 symbol. The PDF417 encoder provides a few API functions to make the rendering process simple. The process is done in two steps. First, a query API is called to calculate size requirements. Second, a render API is called with the values received in the query and the required output callback function. The actual data is returned from the renderer via the callback routine.

Most of the rendering routines use a PDFRENDER structure to control the output. This structure is defined as follows:

```

struct sPdfRenderInfo
{
    unsigned int    modWidth;
    unsigned int    modHeight;
    unsigned int    shaveWidth;
    unsigned int    shaveHeight;
    PDFBOOL bIncludeQuietZones;
};

```

The members of this structure are:

<code>modWidth</code>	Width of each output module in pixels.
<code>modHeight</code>	Height of each output module in pixels.
<code>shaveWidth</code>	Number of pixels to “shave” off each black module horizontally.
<code>shaveHeight</code>	Number of pixels to “shave” off each black module vertically.
<code>bIncludeQuietZones</code>	If <code>PTRUE</code> , the output image will include the required white border around the symbol. If <code>PFALSE</code> , only the symbol itself will be in the output image.

During most printing processes, there is a tendency for the ink or toner to spread slightly. When you print a PDF417 symbol, the black modules tend to end up slightly wider than expected and the white modules slightly narrower. The encoder allows you to compensate for this ink spread by “shaving” the black modules. Choosing a non-zero value for the horizontal “shaving” amount, for example, causes the encoder to lay the symbol out using the specified module width but change part of each black module back to white.

The amount of shaving required depends entirely on the printing process and the type of stock being used. With laser printers, this effect is minimal and can usually be ignored. With inkjet printers, particularly on low-quality paper stock, the amount of ink spread can be significant. You may need to experiment to determine an appropriate setting.

Each of the rendering functions requires a callback function that is used to pass graphics data in the required manner. Provided in the distribution are three sample callback functions as follows;

- `PdfSinkStream` (defined in `sinkfile.c`) – will create a file (`fopen`) and write the graphics data to it.
- `PdfSinkMemory` (defined in `sinkmem.c`) – will write the graphics data to memory.
- `PdfSinkFd` (defined in `sinkfile.c`) – will create a file (`creat`) and write the graphics data to it.

Any callback function can be used as output for the graphics data. It must, however, conform to the callback function prototype discussed later in this section.

5.4 Rendering Callback Routines

5.4.1 *Output Callback Functions*

Description:

The PDF417 encoder requires that you provide a callback function that it uses to pass graphics data to you. This function must conform to the prototype and behavior discussed below.

Prototype:

```
PDFERROR PDFPTR CallbackFunction( const unsigned char PDFPTR *pData,
                                  unsigned int nBytes,
```

```
void PDFPTR *pUser);
```

Arguments:

pData	a pointer to the data stream to write.
nBytes	number of bytes to write.
pUser	a pointer to the output object.

The meaning of the `pUser` parameter depends on the way the particular callback function operates. When a rendering routine is called, it is passed a pointer to the callback function and a user pointer. The user pointer is passed unaltered to the callback function each time the callback function is called.

The PDF417 encoder provides source code for three pre-written callback functions as follows:

<code>PdfSinkMemory</code>	This callback writes data out to memory. When the rendering routine is called, <code>pUser</code> must be the address of the memory pointer, not the memory pointer itself. The pointer will be advanced, so you may need to work on a copy.
<code>PdfSinkStream</code>	This callback writes data out to a file via a <code>FILE</code> pointer. When the rendering routine is called, <code>pUser</code> must be the <code>FILE</code> pointer itself since the prototype is a <code>void PDFPTR *</code> .
<code>PdfSinkFd</code>	This callback writes data out to a file via a file descriptor. When the rendering routine is called, <code>pUser</code> must be the address of the file descriptor.

5.4.2 PdfSinkMemory

Example:

```
#include "pdfenc.h"
#include "pdfrend.h"
#include "sinkmem.h"

void main(void)
{
    PDFSYMBOL mySymbol;
    PDFRENDER myRenderInfo;
    unsigned long lineBufSize;
    unsigned long totalSize;
    char pLineBuf; /* allocated renderer working memory */
    char *pMem, *pCopy;
    PERERROR returnStat = PERR_OK;

    . . . . .

    returnStat = PdfDIBQuery( &mySymbol,
                             &myRenderInfo,
                             PTRUE,
    /* include file header */
                             &totalSize,
                             &lineBufSize
                             );
    if( returnStat != PERR_OK )
    {
```

```

        printf("*** ERROR calling PdfDIBQuery() status = %d", returnStat);
    }

    pLineBuf = malloc( lineBufSize );
    if( pLineBuf == PDFNULL )
    {
        printf("*** ERROR - NO MEMORY");
        exit(0);
    }

    pMem = malloc( totalSize );
    if( pMem == PDFNULL )
    {
        printf("*** ERROR - NO MEMORY");
        exit(0);
    }

    pCopy = pMem;
    returnStat = PdfDIBRender( &mySymbol,
                              &myRenderInfo,
                              PTRUE, /* include file header          */
                              pLineBuf,
                              PdfSinkMemory, /* callback function      */
                              &pCopy); /* pUser                  */

    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfDIBRender() status = %d", returnStat);
    }

    free(pLineBuf);

    . . . . .

    free(pMem);
}

```

5.4.3 PdfSinkStream

Example:

```

#include "pefenc.h"
#include "pdfrend.h"
#include "sinkfile.h"

void main(void)
{
    PDFSYMBOL mySymbol;
    PDFRENDER myRenderInfo;
    unsigned long lineBufSize;
    char pLineBuf;          /* allocated renderer working memory */
    FILE *fp;
    PERERROR returnStat = PERR_OK;

    . . . . .

    fp = fopen("output.bmp", "w");
    if( fp == PEFNULL )
    {
        printf("*** ERROR Opening output.bmp");
        exit(0);
    }

    . . . . .

```

```

returnStat = PdfDIBQuery( &mySymbol,
                        &myRenderInfo,
                        PTRUE,          /* include file header */
                        PDFNULL,
                        &lineBufSize);
if( returnStat != PERR_OK )
{
    printf("*** ERROR calling PdfDIBQuery() status = %d", returnStat);
}

pLineBuf = malloc( lineBufSize );
if( pLineBuf == PDFNULL )
{
    printf("*** ERROR - NO MEMORY");
    exit(0);
}

returnStat = PdfDIBRender( &mySymbol,
                          &myRenderInfo,
                          PTRUE,          /* include file header */
                          pLineBuf,
                          PdfSinkStream, /* callback function */
                          fp); /* pUser */

if( returnStat != PERR_OK )
{
    printf("*** ERROR calling PdfDIBRender() status = %d", returnStat);
}

free(pLineBuf);
fclose(fp);

. . . . .
}

```

5.4.4 PdfSinkFd

Example:

```

#include "pdfenc.h"
#include "pdfrend.h"
#include "sinkfile.h"

void main(void)
{
    PDFSYMBOL mySymbol;
    PDFRENDER myRenderInfo;
    unsigned long lineBufSize;
    char pLineBuf;          /* allocated renderer working memory */
    int fd;
    PERERROR returnStat = PERR_OK;

    . . . . .

    fd = creat("output.bmp", S_IWRITE | O_CREAT);
    if( fd == -1 )
    {
        printf("*** ERROR Opening output.bmp errno = %d", errno);
    }

    . . . . .
}

```

```

returnStat = PdfDIBQuery( &mySymbol,
                        &myRenderInfo,
                        PTRUE,          /* include file header */
                        PDFNULL,
                        &lineBufSize);
if( returnStat != PERR_OK )
{
    printf("*** ERROR calling PdfDIBQuery() status = %d", returnStat);
}

pLineBuf = malloc( lineBufSize );
if( pLineBuf == PDFNULL )
{
    printf("*** ERROR - NO MEMORY");
    exit(0);
}

returnStat = PdfDIBRender( &mySymbol,
                          &myRenderInfo,
                          PTRUE, /* include file header */
                          pLineBuf,
                          PdfSinkFd, /* callback function */
                          &fd); /* pUser */
if( returnStat != PERR_OK )
{
    printf("*** ERROR calling PdfDIBRender() status = %d", returnStat);
}

free(pLineBuf);
close(fd);

. . . . .
}

```

5.5 Device-Independent Bitmap Rendering Routines

5.5.1 PdfDIBQuery

Description:

This function returns the total number of bytes the rendered symbol will require, as well as the number of bytes to render a single "line" of the symbol.

The purpose of this API function is to allow for a dynamic environment for rendering. The symbol may vary significantly depending on how much data is to be encoded and the rendering size requirements. If the user is rendering multiple symbols to fit in different printable areas, this API makes it easier to determine the actual amount of memory required to render the symbol. Also, if memory is a critical issue, you can either render one line of the symbol at a time or all of it at once.

Prototype:

```

#include "pdfrend.h"

PDFERROR PDFAPI PdfDIBQuery( PDFSYMBOLCPTR pSymbol,
                            PDFRENDERCPTR pRenderInfo,
                            PDFBOOL bIncludeFileHdr,
                            unsigned long PDFPTR *pTotalSize,
                            unsigned long PDFPTR *pLineBufSize );

```

Arguments:

<code>pSymbol</code>	a pointer to a <code>PDFSYMBOL</code> structure.
<code>pRenderInfo</code>	a pointer to a <code>PDFRENDER</code> structure (defined in <code>pdfrend.h</code>).
<code>bIncludeFileHdr</code>	Boolean value indicated to include the <code>BITMAPFILEHEADER</code> header information when creating the output. If the output is going to a file, this should be <code>PTRUE</code> . If it is going to memory, it should probably be <code>PFALSE</code> .
<code>pTotalSize</code>	a pointer to a variable that will receive the total size in bytes of the DIB representing the PDF417 symbol. This may be <code>NULL</code> if this information is not required.
<code>pLineBufSize</code>	a pointer to a variable that will receive the size in bytes of the temporary line buffer required by the renderer. This may be <code>NULL</code> if this information is not required.

NOTES:

- The include file `pdfrend.h` contains all of the type definitions and prototypes for the render API functions.
- Either `pTotalSize` or `pLineBufSize` may be `NULL`, but not both.
- The value output through `pLineBufSize` represents the minimum size of the line buffer that will be needed by `PdfDIBRender`.

5.5.2 PdfDIBRender:

Description:

This function performs the actual formatting of the DIB image.

Prototype:

```
#include "pdfrend.h"

PDFERROR PDFAPI PdfDIBRender( PDFSYMBOLCPTR pSymbol,
                               PDFRENDERCPTR pRenderInfo,
                               PDFBOOL bIncludeFileHdr,
                               unsigned char PDFPTR *pLineBuf,
                               PDFDATASINK pSinkFcn,
                               void PDFPTR *pUser );
```

Arguments:

pSymbol	a pointer to a PDFSYMBOL structure.
pRenderInfo	a pointer to a PDFRENDER structure (defined in pdfrend.h)
bIncludeFileHdr	Boolean value indicated to include the BITMAPFILEHEADER header information when creating the output. If the output is going to a file, this should be PTRUE. If it is going to memory, it should probably be PFALSE.
pLineBuf	a pointer to an area of memory that PdfDIBRender can use as working storage. The minimum size of this buffer can be obtained from PdfDIBQuery.
pSinkFcn	a pointer to the output callback function that will be called to pass the data out.
pUser	user pointer that will be passed to the callback function.

NOTES:

- The include file pdfrend.h contains all of the type definitions and prototypes for the render API functions.
- Refer to PdfDIBQuery for further documentation on structures.
- This function assumes that the memory allocated for pLineBuf is large enough. No error checking is performed in the function for memory overwriting.
- If the graphics data is going to be output to a file and then used, the operating system needs to know the type of file it is. To make sure that the file header is written to the output file, set bIncludeFileHdr to PTRUE. If the file will not be used as an actual bitmap file, no file header is required and this value can be set to PFALSE.

Example:

```
#include "pdfenc.h"
#include "pdfrend.h"
```

```
#include "sinkfile.h"

void main(void)
{
    PDFSYMBOL userSymbol;
    PDFRENDERINFO renderInfo;
    PDFBOOL bIncludeFileHdr;
    unsigned long lineBufSize;
    unsigned char *pLineBuf;
    FILE *fout;
    char outFile[15];
    PDFERROR returnStat = PERR_OK;

    . . . . .

    renderInfo.modWidth = (unsigned int) aspectRatio.moduleWidth;
    renderInfo.modHeight = (unsigned int) aspectRatio.moduleHeight;
    renderInfo.shaveWidth = 0;
    renderInfo.shaveHeight = 0;
    renderInfo.bIncludeQuietZones = PTRUE;

    /*
     * Since routine is writing to a file, the header
     * information is required.
     */
    bIncludeFileHdr = PTRUE;

    returnStat = PdfDIBQuery( &userSymbol,
                             &renderInfo,
                             bIncludeFileHdr,
                             PDFNULL,
                             &lineBufSize );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfDIBQuery() status = %d", returnStat);
    }

    pLineBuf = malloc( lineBufSize );
    if( pLineBuf == PDFNULL )
    {
        printf("*** ERROR - NO MEMORY");
        exit(0);
    }

    strcpy(outFile, "output.bmp");
    fout = fopen(outFile, "w");
    if( fout == PDFNULL )
    {
        printf("*** ERROR Opening %s", outFile);
        exit(0);
    }

    returnStat = PdfDIBRender( &userSymbol,
                              &renderInfo,
                              bIncludeFileHdr,
                              pLineBuf,
                              PdfSinkStream,
                              fout );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfDIBRender() status = %d", returnStat);
    }
}
```

```

        fclose(fout);
        free(pLineBuf);

        . . . . .
    }

```

5.6 Tagged Image File Format (TIFF) Rendering Routines

5.6.1 *PdfTIFFQuery*:

Description:

This function returns the total number of bytes the rendered symbol will require, as well as the number of bytes to render a single "line" of the symbol.

The purpose of this API function is to allow for a dynamic environment for rendering. The symbol may vary significantly depending on how much data is to be encoded and the rendering size requirements. If the user is rendering multiple symbols to fit in different printable areas, this API makes it easier to determine the actual amount of memory required to render the symbol. If memory is a critical issue, you can either render one line of the symbol at a time or all of it at once.

Prototype:

```

#include "pdfrend.h"

PDFERROR PDFAPI PdfTIFFQuery( PDFSYMBOLCPTR pSymbol,
                              PDFRENDERCPTR pRenderInfo,
                              unsigned long PDFPTR *pTotalSize,
                              unsigned long PDFPTR *pLineBufSize );

```

Arguments:

`pSymbol` a pointer to a PDFSYMBOL structure.
`pRenderInfo` a pointer to a PDFRENDER structure (defined in pdfrend.h).
`pTotalSize` a pointer to a variable that will receive the total size in bytes of the TIFF representing the PDF417 symbol. This may be NULL if this information is not required.
`pLineBufSize` a pointer to a variable that will receive the size in bytes of the temporary line buffer required by the renderer. This may be NULL if this information is not required.

NOTES:

- The include file `pdfrend.h` contains all of the type definitions and prototypes for the render API functions.
- Either `pTotalSize` or `pLineBufSize` may be NULL, but not both.
- The value output through `pLineBufSize` represents the minimum size of the line buffer that will be needed by `PdfTIFFRender`.
- This function assumes that the memory allocated for `pLineBuf Size` is large enough. No error checking is performed in the function for memory overwriting.

5.6.2 PdfTIFFRender:

Description:

This function performs the actual formatting of the TIFF image.

Prototype:

```
#include "pdfrend.h"

PDFERROR PDFAPI PdfTIFFRender( PDFSYMBOLCPTR pSymbol,
                                PDFRENDERCPTR pRenderInfo,
                                PDFTIFFCPTR pTiffInfo,
                                unsigned char PDFPTR *pLineBuf,
                                PDFDATASINK pSinkFcn,
                                void PDFPTR *pUser );
```

Arguments:

pSymbol a pointer to a PDFSYMBOL structure.
 pRenderInfo a pointer to a PDFRENDER structure (defined in pdfrend.h).
 pTiffInfo a pointer to a PDFTIFF structure (defined in pdfrend.h). This structure contains TIFF-specific information.
 pLineBuf a pointer to an area of memory that PdfTIFFRender can use as working storage. The minimum size of this buffer can be obtained from PdfTIFFQuery.
 pSinkFcn a pointer to the output callback function that will be called to pass the data out.
 pUser a user pointer that will be passed to the callback function.

A PDFTIFF structure is defined as follows:

```
struct sPdfTIFFInfo
{
    unsigned long    xRes;    /* Desired x resolution. */
    unsigned long    yRes;    /* Desired y resolution. */
    int              byteOrder; /* Byte order used within the TIFF file. */
};
typedef struct sPdfTIFFInfo PDFTIFF;
typedef struct sPdfTIFFInfo PDFPTR *PDFTIFFPTR;
typedef const struct sPdfTIFFInfo PDFPTR *PDFTIFFCPTR;
```

The members of this structure are as follows:

xRes	Horizontal resolution (in pixels per inch) to be indicated in the file.
yRes	Vertical resolution (in pixels per inch) to be indicated in the file.
byteOrder	The byte order to use when creating the file. The TIFF file format allows the user of either little-endian (Intel) or big-endian (Motorola) format. This member must be set to BYTE_ORDER_LE to specify Intel format or BYTE_ORDER_BE to specify Motorola format. Intel format is typically used on PC's, while Motorola format is typically used in Macintosh systems.

NOTES:

- The include file pdfrend.h contains all of the type definitions and prototypes for the render API functions.
- Refer to PdfTIFFQuery for further documentation on structures.

Example:

```

#include "pdfenc.h"
#include "pdfrend.h"
#include "sinkfile.h"

void main(void)
{
    PDFSYMBOL userSymbol;
    PDFRENDERINFO renderInfo;
    PDFBOOL bIncludeFileHdr;
    unsigned long lineBufSize;
    unsigned char *pLineBuf;
    FILE *fout;
    char outFile[15];
    PDFERROR returnStat = PERR_OK;

    . . . . .

    renderInfo.modWidth = (unsigned int) aspectRatio.moduleWidth;
    renderInfo.modHeight = (unsigned int) aspectRatio.moduleHeight;
    renderInfo.shaveWidth = 0;
    renderInfo.shaveHeight = 0;
    renderInfo.bIncludeQuietZones = PTRUE;

    /*
     * Since routine is writing to a file, the header
     * information is required.
     */
    bIncludeFileHdr = PTRUE;

    returnStat = PdfTIFFQuery( &userSymbol,
                              &renderInfo,
                              bIncludeFileHdr,
                              PDFNULL,
                              &lineBufSize );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfTIFFQuery() status = %d", returnStat);
    }

    pLineBuf = malloc( lineBufSize );
    if( pLineBuf == PDFNULL )
    {
        printf("*** ERROR - NO MEMORY");
        exit(0);
    }

    strcpy(outFile, "output.tif");
    fout = fopen(outFile, "w");
    if( fout == PDFNULL )
    {
        printf("*** ERROR Opening %s", outFile);
        exit(0);
    }

    returnStat = PdfTIFFRender( &userSymbol,
                                &renderInfo,
                                bIncludeFileHdr,
                                pLineBuf,
                                PdfSinkStream,
                                fout );

    if( returnStat != PERR_OK )

```

```
{
    printf("*** ERROR calling PdfTIFFRender() status = %d", returnStat);
}

fclose(fout);
free(pLineBuf);
}
```

5.7 Font Based Rendering Routines

5.7.1 *PdfFontInitRender*

Description:

This function gives the user the ability to re-map the resultant font characters.

The purpose of this API function is to allow the user to change the font character set to map to any font definition file. It is not a required function as the default font character set is loaded at runtime and will most likely work for most environments.

The NOTE below in the sample code is a warning to be careful as to what buffer size (width & height) you choose to put the symbol font characters into. If the size calculations result in a fractional part then the symbol could contain a partial rendering with a font character which would result in the symbol being improperly generated. A check is performed to insure that the font characters that are generated will fit into the receiving area. An error code of 44 is issued when the font characters will not fit into the symbol and the font characters are not moved into the receiving area. To get the complete symbol you would exceed the maximum area for rendering the symbol thus the instructions for rounding down so as NOT to exceed the required area for the symbol to render to.

Prototype:

```
#include "pdfrend.h"

PDFERROR PDFAPI PdfFontInitRender( const unsigned char PDFPTR *pCharSet );
```

Arguments:

pCharSet a pointer to an unsigned character array which contains the new font character set.

NOTES:

- The include file `pdfrend.h` contains all of the type definitions and prototypes for the render API functions.

5.7.2 PdfFontRender:

Description:

This function performs the actual output of the font characters to an output buffer. The function will render the symbol to a font character set which can then be sent to a printer with the font definition loaded which prints the appropriate bar, space pattern. There is one issue that is important here and will need to be understood for expected behavior. The output buffer is the area where the symbol's font characters are placed with whatever alignment is chosen. It is not the determining factor for the PDF417 symbol shape. Be aware that the symbol shape has already been determined by the setting of the user parameters. The symbol generated must fit into the output buffer or an error will occur. An error code of 44 is issued when the font characters required to render the symbol will not fit into the output buffer. The font characters are not moved into the output buffer and a space filled buffer is returned. This error occurs when the number of rows or columns of the output buffer is smaller than the number of rows and columns needed to render the symbol.

Prototype:

```
#include "pdfrend.h"

PDFERROR PDFAPI PdfFontRender( PDFSYMBOLCPTR pSymbol,
                               PDFRENDERCPTR pRenderInfo,
                               PDFBUFFERINFOCPTR pBuffInInfo,
                               PDFBUFFERINFOPTR pBuffOutInfo,
                               char PDFPTR *pOutputBuffer );
```

Arguments:

pSymbol	a pointer to a PDFSYMBOL structure.
pRenderInfo	a pointer to a PDFRENDER structure (defined in pdfrend.h).
pBuffInInfo	a pointer to a PDFBUFFERINFO structure (defined in pdfrend.h).
pBuffOutInfo	a pointer to a PDFBUFFERINFO structure (defined in pdfrend.h).
pOutputBuffer	a pointer to character array to receive the font characters.

A PDFBUFFERINFO structure is defined as follows:

```
struct sPdfBufferInfo
{
    unsigned short    nWidth;
    unsigned short    nHeight;
    unsigned char     horizAlign;
    unsigned char     vertAlign;
};
typedef struct sPdfBufferInfo PDFBUFFERINFO;
typedef struct sPdfBufferInfo PDFPTR *PDFBUFFERINFOPTR;
typedef const struct sPdfBufferInfo PDFPTR *PDFBUFFERINFOCPTR;
```

The members of this structure are as follows:

nWidth	Width of the buffer (# of columns of font characters). Note that the user parameters <code>minCols</code> and <code>maxCols</code> , used to calculate the actual number of symbol columns set for the symbol shape must fit into this value. Symbol columns multiplied by 4 is the number of font character columns required to render the symbol.
nHeight	Height of the buffer (# of rows of font characters). Note that the user parameters, <code>minRows</code> and <code>maxRows</code> , used to calculate the actual number of symbol rows set for the symbol shape must fit into this value.
horizAlign	The required horizontal alignment of the font characters within a single row. The choices are (L)eft, (C)enter, or (R)ight. The default value is Left.
vertAlign	The required vertical alignment of the rows in the buffer. The choices are (T)op, (C)enter, or (B)ottom. The default value is Top.

NOTES::

- The include file `pdfrend.h` contains all of the type definitions and prototypes for the render API functions.
- The font characters are positioned in the output buffer according to the horizontal and vertical requirements. If the output buffer is larger than the generated font character array then it is filled with the space font character.

Example:

```
#include "pdfenc.h"
#include "pdfrend.h"

void main(void)
{
    char fontCharSet[16];
    PDFSYMBOL userSymbol;
    PDFRENDERINFO renderInfo;
    PDFBUFFININFO buffInInfo, buffOutInfo;
    unsigned int fontRows, fontCols;
    char PDFPTR *pBuffer;
    PDFERROR returnStat = PERR_OK;

    . . . . .

    /*
     * The setting of the font character array is NOT required.
     * The following is an example of setting the font character set,
     * if the default character set does not work with your font definition.
     * There are 16 code points defined for the default font.
     */
    strncpy(fontCharSet, " ABCDEFGHIJKLMNO", 16);
    returnStat = PdfFontInitRender( fontCharSet );
    if( returnStat != PERR_OK )
    {
        printf("*** ERROR calling PdfFontInitRender() status = %d",
              returnStat);
    }

    . . . . .

    renderInfo.modWidth = (unsigned int) aspectRatio.moduleWidth;
```

```

renderInfo.modHeight = (unsigned int) aspectRatio.moduleHeight;
renderInfo.shaveWidth = 0;
renderInfo.shaveHeight = 0;
renderInfo.bIncludeQuietZones = PFALSE;

/*
 * Allocate enough memory to hold font characters in the output
 * buffer.
 *
 * The output area will need to be calculated by using the area
 * in inches then converting to pixels and finally converting
 * to characters. The API will convert to symbol rows and columns
 * internally and fill the unused area with spaces. The print
 * routine will print the entire area.
 *
 * e.g.:
 *      Width = 1.50 inches
 *      Height = 0.75 inches
 *
 * To convert to pixels with 300 dpi:
 *      300 * 1.5 = 450 pixels
 *      300 * .75 = 225 pixels
 *
 * To convert to chars from pixels with a 10 millage font definition
 * of 12 pixels per character width and 9 pixels per character height:
 *      450 / 12 = 37.5 characters in 1 row
 *      225 / 9 = 25 character rows
 *
 * NOTE: Round down the values because the guarantee is that the symbol
 * will NOT exceed this area when it is set with font characters.
 *
 *      Width = 37 characters maximum
 *      Height = 25 characters maximum
 */
fontCols = 37;
fontRows = 25;

pBuffer = (char *) malloc( sizeof(PDFFONTOUTCOBOL) +
                          ( fontRows * fontCols));
if( pBuffer == PDFNULL )
{
    printf("\n***** NO MEMORY - malloc() *****");
    return;
}

buffInInfo.nWidth = (unsigned short) fontCols;
buffInInfo.nHeight = (unsigned short) fontRows;
buffInInfo.horizAlign = 'R'; /* Align right horizontally */
buffInInfo.vertAlign = 'B'; /* Align bottom vertically */

/*
 * Upon return from PdfFontRender(), the output buffer contains the
 * font characters which, if sent to a printer with the appropriate
 * font definition loaded, will print the PDF417 symbol.
 */
returnStat = PdfFontRender( &userSymbol,
                          &renderInfo,
                          &buffInInfo,
                          &buffOutInfo,
                          pBuffer );

if( returnStat != PERR_OK )
{
    printf("*** ERROR calling PdfFontRender() status = %d", returnStat);
}

```

```

    }
}

```

5.8 EBCDIC-to-ASCII and Utility Routines

For most implementations of the C API, you will be using the ASCII character set for input data, however, there may be cases where you want to implement the PDF417 Encoder in C on mainframe or mid-range systems which use the EBCDIC character set. The encoder library provides facilities to handle EBCDIC-to-ASCII conversions automatically for the internal data structures within the C API. With regard to the input data to be encoded, the conversion decision is left up to the user. To accommodate this possibility, there are API's which are available for translating the EBCDIC data to ASCII.

5.8.1 *PdfEtoA*

Description:

This function performs an unconditional conversion of EBCDIC character data to ASCII character data.

Prototype:

```

#include "pdfenc.h"

PDFERROR PDFAPI PdfEtoA( void PDFPTR *pDest,
                        const void PDFPTR *pSrc,
                        unsigned int numChars );

```

Arguments:

pDest	a pointer to the destination area.
pSrc	a pointer to the source area.
numChars	the number of characters to translate.

Example:

```

#include "pdfenc.h"

void main(void)
{
    unsigned int nBytes;
    char inData[200];
    unsigned char aData[200];
    unsigned char *pData;

    . . . . .

    strcpy(inData,"This is a test of the Silver Bay Software"
           " PDF417 Encoder. The error correction will encode at "
           "level 3 with ECC padding.");
    nBytes = strlen(inData);

    /*
     * When executing this in an EBCDIC environment then need to
     * convert the data to ASCII.
     */
    pData = aData;
    PdfSet(pData,'\0', nBytes);
}

```

```

    PdfEtoA(pData, inData, nBytes);
    . . . . .
}

```

5.8.2 PdfSet

Description:

This function is a substitute for memset(). There is no formal requirement of having the C development environment when using the encoder so the API provides this functionality.

Prototype:

```

#include "pdfenc.h"

PDFERROR PDFAPI PdfSet( void PDFPTR *pBytes,
                        int c,
                        unsigned int nBytes );

```

Arguments:

pBytes	a pointer to the area of memory to set.
c	the character to set.
nBytes	the number of bytes to set.

NOTES:

- If a NULL is passed in for the pointer to set, no action is performed.

Example:

```

#include "pdfenc.h"

void main(void)
{
    unsigned int nBytes;
    char inData[200];
    unsigned char aData[200];
    unsigned char *pData;

    . . . . .

    strcpy(inData, "This is a test of the Silver Bay Software"
           " PDF417 Encoder.");
    nBytes = strlen(inData);

    /*
     * When executing this in an EBCDIC environment then need to
     * convert the data to ASCII.
     */
    pData = aData;
    PdfSet(pData, '\0', nBytes);

    . . . . .
}

```

6 COBOL Language API

In order to simplify the programming interface, the COBOL API is somewhat less flexible than the C API. In particular, many of the steps that are performed individually in the C API are combined into single operations in the COBOL API.

6.1.1 Overview of the Encode Process

There are essentially four steps to using the encoder. Briefly, they are as follows:

1. Font information initialization.
2. Encoder parameter initialization.
3. Perform the data encode.
4. Print the symbol. This step is the user's responsibility, and is not performed with functions provided by the encoder API.

If additional symbols need to be generated, the process may be repeated starting with either Step 2 (if the parameters need to be modified) or Step 3 (if the parameters do not need to be changed).

Each of these steps is described in more detail below. Manual pages for the individual API subroutines are provided at the end of this section.

6.1.2 Font Information Initialization

The COBOL API for the encoder is restricted to font rendering. Graphics rendering is not supported.

Before the encoding API function is called, the encoder might require initialization. This process indicates to the encoder which font character values it should use to output the encoded symbol. This is important because different printers result in different character translations between host and printer. The PDF417 encoder compensates for such variations by outputting different values, by using different font files, or both. Be sure you read Section 9 for information related to how to print the symbol characters properly in your environment.

The font initialization function does not need to be called before every symbol encoded. It only needs to be called once as part of program initialization. If this function is not called, the default values indicated in the API definition are used.

6.1.3 Encoder Parameters Initialization

As described in Section 3, there are a variety of control parameters that will affect how a PDF417 symbol is generated. The COBOL API allows to programmer to provide this information to the encoder using a record formatted as shown below:

```

01 PDF417-PARAMETER-INFO-REC.
   03 PDF-VERSION                PIC 9(4)    VALUE 200.
   03 NUM-BYTES-TO-ENCODE        PIC 9(4)    VALUE 0.
   03 ASPECT-RATIO.
      05 SYMBOL-HEIGHT           PIC 9(2)    VALUE 1.
      05 SYMBOL-WIDTH            PIC 9(2)    VALUE 2.
   03 ECC-TYPE                   PIC 9(1)    VALUE 1.
   03 ECC-VALUE                  PIC 9(3)    VALUE 5.
   03 NO-PAD-ECC-FLAG           PIC X(1)    VALUE 'F'.
   03 INPUT-DATA-FORMAT          PIC X(1)    VALUE 'N'.
   03 SYMBOL-TYPE                PIC X(1)    VALUE 'S'.
   03 ENCODE-METHOD           PIC X(1)    VALUE 'O'.
   03 PRINT-SYMBOL-HORIZ-ALIGN   PIC X(1)    VALUE 'L'.
   03 PRINT-SYMBOL-VERT-ALIGN   PIC X(1)    VALUE 'T'.
   03 EBCDIC-TO-ASCII-FLAG      PIC X(1)    VALUE 'Y'.

```

The table below summarizes the individual parameters. Certain of them are then discussed in more detail after the table.

Item	Meaning / Usage
PDF-VERSION	This field is used for forward compatibility, in case the format of the PDF417-PARAMETER-INFO-REC changes in the future. As of this writing, this field must be set to a value of 200, indicating the 2.x data format above.
NUM-BYTES-TO-ENCODE	This value is set with the number of characters of input data that are to be encoded. This must match the length of the data set passed to the PDFENCOD procedure.
ASPECT-RATIO. SYMBOL-HEIGHT SYMBOL-WIDTH	These values control the preferred aspect ratio of the output symbol. See Section 3.2.5 for more details on aspect ratio.
ECC-TYPE	How the error correction is being specified. There are three choices: <ul style="list-style-type: none"> 0 Use the AIM recommended defaults. In this case, the encoder will automatically follow the recommendations in the PDF417 symbology standard. NOTE: If the this value is set, the ECC-VALUE parameter will be ignored 1 Use a specific error correction level. In this case, the encoder will use the specific error correction level you specify in the ECC-VALUE parameter. 2 Select the ECC level as a percentage of the data codeword count. If you select this option, you also specify a percent in the ECC-VALUE parameter. The encoder will choose the lowest level of error correction that includes at least this percentage of ECC codewords. For example, if you specify 10%, and the symbol has 250 codewords, the encoder will include at least 25 error correction codewords. Since one of the levels below must be chosen, the encoder will, in this case, use level 4, which includes 32 error correction codewords.

Item	Meaning / Usage
ECC-VALUE	<p>The value of this parameter depends on the value of the ECC-TYPE parameter:</p> <p>ECC-TYPE = 0 In this case, this value is ignored and should be set to zero.</p> <p>ECC-TYPE = 1 This value encodes the ECC level (1-8) that should be used when constructing the symbol.</p> <p>ECC-TYPE = 2 This value contains the percentage of codewords that should be ECC codewords.</p>
NO-PAD-ECC-FLAG	<p>Whether to suppress the default “pad with ECC” behavior of the encoder. (See Section 3.2.6)</p> <p>‘T’ Padding will <i>not</i> be done using ECC codewords.</p> <p>‘F’ Padding <i>will</i> be done using ECC codewords. (This is the recommended setting).</p> <p>The default value is ‘F’.</p>
INPUT-DATA-FORMAT	<p>How the input data format is being specified. (See Section 3.3.) There are three choices:</p> <p>‘N’ None default (No escape sequences present in data).</p> <p>‘E’ ECI format (Extended Channel Interpretations).</p> <p>‘G’ GLI format (Global Label Identifier).</p> <p>The default value is ‘N’.</p>
SYMBOL-TYPE	<p>What type of symbol will be generated. (See Section 2.4) There are two choices:</p> <p>‘S’ A standard PDF417 symbol will be generated.</p> <p>‘T’ The encoder will generate a “Truncated” symbol – one that lacks the right row indicators and stop pattern.</p> <p>The default value is ‘S’.</p>
ENCODE-METHOD	<p>How the input data will be encoded. (See Section 3.1) There are two choices:</p> <p>‘O’ Optimal.</p> <p>‘B’ Binary.</p> <p>The default value is ‘O’.</p>
PRINT-SYMBOL-HORIZ-ALIGN	<p>How the font characters will be horizontally aligned in the output area. There are three choices:</p> <p>‘L’ Left</p> <p>‘C’ Center</p> <p>‘R’ Right</p> <p>The default value is ‘L’</p>
PRINT-SYMBOL-VERT-ALIGN	<p>How the font characters will be vertically aligned in the output table. There are three choices:</p> <p>‘T’ Top</p> <p>‘C’ Center</p> <p>‘B’ Bottom</p> <p>The default value is ‘T’.</p>

Item	Meaning / Usage
EBCDIC-TO-ASCII-FLAG	Indicates if the encoder should treat the input data as EBCDIC and convert it to ASCII. There are two choices: 'T' True 'F' False The default value is 'T'.

6.1.4 COBOL Output record Initialization

The output of the PDF417 encoder is a table of characters which, when printed using the provided custom PDF417 font, renders a PDF417 symbol. The actual size of this table can vary greatly, as it's based on the amount of data being encoded, the ECC level being used, the aspect ratio of the symbol, and the font being used.

As the application programmer, you must determine the appropriate size of this table, declare it in your application program, and initialize it correctly.

The general structure of the output record is as follows:

```

01 PDF417-OUTPUT-REC.
   05 PRINT-SYMBOL-WIDTH      PIC 9(3) VALUE W.
   05 PRINT-SYMBOL-HEIGHT     PIC 9(3) VALUE R.
   05 RESULT-CODE             PIC 9(3) VALUE 0.
   05 OUTPUT-LINES           PIC X(W) OCCURS R.

```

where **W** is the width of each row in the table and **R** is the number of rows in the table. The PRINT-SYMBOL-WIDTH, PRINT-SYMBOL-HEIGHT, and RESULT-CODE fields are always required and must be declared as PIC (3) fields. The OUTPUT-LINES field is also required; however, its size must be determined by the programmers, and then specified in the PRINT-SYMBOL-WIDTH and PRINT-SYMBOL-HEIGHT fields. For example, if we required 8 lines of output, each 43 characters long, then the output record structure would be declared and initialized as follows:

```

01 PDF417-OUTPUT-REC.
   05 PRINT-SYMBOL-WIDTH      PIC 9(3) VALUE 43.
   05 PRINT-SYMBOL-HEIGHT     PIC 9(3) VALUE 8.
   05 RESULT-CODE             PIC 9(3) VALUE 0.
   05 OUTPUT-LINES           PIC X(43) OCCURS 8.

```

The important point here is that PRINT-SYMBOL-WIDTH and PRINT-SYMBOL-HEIGHT must precisely reflect the size of the OUTPUT-LINES table that has been declared in the record.

Note that it is not strictly required that a table be used to declare the output lines. In certain circumstances, it may be more convenient to the application program to have each of the output lines of the encoder in a separate field (for example, this makes interfacing with a printer DDS on an AS/400 simpler). The following record could have been used as it represents exactly the same storage:

```

01 PDF417-OUTPUT-REC.
   05 PRINT-SYMBOL-WIDTH      PIC 9(3) VALUE 43.
   05 PRINT-SYMBOL-HEIGHT     PIC 9(3) VALUE 8.
   05 RESULT-CODE             PIC 9(3) VALUE 0.
   05 OUTPUT-LINE1           PIC X(43).
   05 OUTPUT-LINE2           PIC X(43).
   05 OUTPUT-LINE3           PIC X(43).

```

```

05 OUTPUT-LINE4      PIC X(43) .
05 OUTPUT-LINE5      PIC X(43) .
05 OUTPUT-LINE6      PIC X(43) .
05 OUTPUT-LINE7      PIC X(43) .
05 OUTPUT-LINE8      PIC X(43) .

```

Determining the appropriate dimension of the output area requires an understanding of the structure of a PDF417 symbol. Please read Section 2.1 and Section 2.2 if you have not already done so.

The font that is used to print PDF417 characters is one module high. Thus, you will require one line in the output area for each row that may appear in the symbol. Each character in the custom font represents four modules horizontally. Thus, for each character you allocate in each output line, you will have room for four modules. Thus, in our example above, the output symbol is limited to 8 rows, and $43 * 4 = 172$ modules. Using the procedure in Section 2.2.2, we can see that the 172 modules corresponds to

```
INTEGER((172 - 73) / 17) = 5
```

data columns if the symbol is to include the white space, and

```
INTEGER((172 - 69) / 17) = 6
```

data columns if the symbol is not to include the white space. Assuming the latter for the moment, this means that the symbol has space for $6*8=48$ total codewords.

The more common need, however, is to work backwards from a physical area on paper to determine what values we should use.

Here is an example: assume we have an area of 1.75 inches by 0.25 inches available to use for printing and we're using the 10 mil Xerox font (a 300 dot per inch font). The font's name is X5P310; thus, from the name we can see that the module width is 3 and the module height is 10. (See Section 9.4 for more information on the fonts.)

A module width of 3 pixels at 300 dpi corresponds to a physical module width of 0.010 inches, while a module height of 10 pixels at 300 dpi corresponds to a physical module height of 0.033 inches. This means that the physical number of modules available in our defined space is:

```

horizontal_modules = space_available / module_width = 1.75 / 0.010 = 175
vertical_modules   = space_available / module_height = 0.25 / 0.033 = 7.5

```

Since each character in the font encodes four modules, this means that we need to allocate $175/4=43.75$ characters horizontally and 7.5 rows to exactly fill this area.

Now obviously we can't declare a COBOL table that is 43.75 by 7.5; we must decide what to do with the decimal portion. There are some subtle and complex implications of rounding these numbers up or down.

Let's consider rounding the number of rows first. If we round the number of row down (to 7) this will guarantee that the symbol's height will never exceed 0.25 inches. The height of the symbol would be the height of 7 rows of this font:

```
symbol_height = num_rows * height_of_module = 7 * 0.0333 = 0.233 inches
```

Thus, if we only allow for 7 rows in our symbol, the maximum possible height is 0.233 inches. If we round the number of rows up (to 8), we are now increasing the size of the symbol:

```
symbol_height = num_rows * height_of_module = 8 * 0.0333 = 0.266 inches
```

Note that the increase in the symbol's height is not dramatic (only 0.033 inches for this particular font).

From this we can surmise that if the printed height you've specified for your symbol *absolutely cannot be exceeded*, then round the number of rows **down**. Conversely, if the print area is flexible (especially considering the relatively low overhead of adding an additional row), then round the number of rows **up**.

The process of rounding the row width is much more complex, as more than just the decimal portion will likely need to be added or subtracted to affect the actual width of the PDF417 symbol. From above you can see that adding or subtracting a single row of characters will change the height of the symbol. However, adding or subtracting a single character to the width may not change the width of the symbol.

Recall from Section 2.2.2 that the number of data columns in a symbol is related to the available number of module widths by the expression

```
data_columns = INTEGER((modules_available - 73) / 17)
```

or

```
data_columns = INTEGER((modules_available - 69) / 17)
```

depending on whether or not the quiet zone is being included in the generated area. Because each character in the font represents four modules, we can convert these to:

```
data_columns = INTEGER((chars_across * 4 - 73) / 17)
```

or

```
data_columns = INTEGER((chars_across * 4 - 69) / 17)
```

Rounding the 43.75 up to 44 would have no effect, as this still only allows 6 data columns:

```
(44 * 4 - 69) / 17 = 6.29
```

In order to increase the number of data columns from 6 to 7, we would have to increase the width of each row to 47 characters.

```
(47 * 4 - 69) / 17 = 7.00
```

Note that unlike adding a single row to the symbol, adding an additional data column can have a much more dramatic impact on the overall width, since each PDF417 data column consists of 17 modules. In our example, the overall width would go from 1.71 inches to 1.84.

6.1.5 Encoding Data

The actual data encoding process is performed using one function. The function is:

PDFENCOD

This function will generate the font characters, given the input data, which can then be passed on to the printer.

Given that the COBOL information record and output record have been initialized appropriately, this function will encode the input data and generate a table of font characters that, when printed with the appropriate font definition, will result in a PDF417 symbol.

6.1.6 Printing the Symbol

Printing the symbol is your (the user's) responsibility. Because the printing requirements vary so much from system to system and application to application, the API provides no functions to perform the printing operation.

6.1.7 Result Codes

Each API function returns a status result value. The COBOL API does not rely on the special register RETURN CODE since this may or may not exist in your COBOL language. Instead, it uses a field in the output record as described above. If the encode and symbol generation is successful, the status will return 0. Refer to Section 10.1 for expected values and detailed definitions.

7 COBOL Language API Functions

This section of the document provides detailed descriptions of each of the COBOL API routines.

7.1.1 PDFINITF

Description:

This function gives the user the ability to re-map the resultant font characters. The API function is provided to allow the user to change the font character set to map to any font definition file. It is not a required function as the default font character set is loaded at runtime and will work for most environments.

The primary situation in which this function might need to be called is in an EBCDIC environment printing to a Xerox printer. The encoder will, by default, use the characters “space” and “A” through “O” in generating the barcode output. On an EBCDIC computer, these will be the EBCDIC values of the characters. The Xerox printer, however, expects to receive the characters in ASCII. Thus, a conversion must be performed somewhere. This can be done by changing the characters used by the encoder, or via some other means.

Call:

```

WORKING STORAGE.
  01 FONT-CHAR-SET.
    03 PDFFONT-SPACE PIC X(1) VALUE ' '.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'A'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'B'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'C'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'D'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'E'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'F'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'G'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'H'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'I'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'J'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'K'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'L'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'M'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'N'.
    03 PDFFONT-CHAR1 PIC X(1) VALUE 'O'.

CALL 'PDFINITF' USING FONT-CHAR-SET.

```

Parameters:

FONT-CHAR-SET a data type (length 16) which contains the new font character set.

NOTES:

- Be sure you read Section 9 and the sections appropriate for your printing environment.
- If you are using the encoder as an AS/400 ILE service program, remember to include LINKAGE TYPE IS PROCEDURE in the call statement. Refer to the AS/400 sample programs.

The parameter used by this function is a data type with 16 elements. These elements are described as follows:

Element	Index	Contents – using 4 module width rectangles
PDFFONT-SPACE	0x00	The “full width blank” character.
PDFFONT-CHAR1	0x01	The character containing the first (from right) rectangle.
PDFFONT-CHAR2	0x02	The character containing the second rectangle.
PDFFONT-CHAR3	0x03	The character containing the first & second rectangle.
PDFFONT-CHAR4	0x04	The character containing the third rectangle.
PDFFONT-CHAR5	0x05	The character containing the third & first rectangle.
PDFFONT-CHAR6	0x06	The character containing the third & second rectangle.
PDFFONT-CHAR7	0x07	The character containing the third, second & first rectangle.
PDFFONT-CHAR8	0x08	The character containing the fourth rectangle.
PDFFONT-CHAR9	0x09	The character containing the fourth & first rectangle.
PDFFONT-CHAR10	0x0A	The character containing the fourth & second rectangle.
PDFFONT-CHAR11	0x0B	The character containing the fourth, second & first rectangle.
PDFFONT-CHAR12	0x0C	The character containing the fourth & third rectangle.
PDFFONT-CHAR13	0x0D	The character containing the fourth, third & first rectangle.
PDFFONT-CHAR14	0x0E	The character containing the fourth, third & second rectangle.
PDFFONT-CHAR15	0x0F	The “full width black” character (All four rectangles).

Default values output by the PDF417 encoder (in hexadecimal):

Element	ASCII	EBCDIC
PDFFONT-SPACE	0x20	0x40
PDFFONT-CHAR1	0x41	0xC1
PDFFONT-CHAR2	0x42	0xC2
PDFFONT-CHAR3	0x43	0xC3
PDFFONT-CHAR4	0x44	0xC4
PDFFONT-CHAR5	0x45	0xC5
PDFFONT-CHAR6	0x46	0xC6
PDFFONT-CHAR7	0x47	0xC7
PDFFONT-CHAR8	0x48	0xC8
PDFFONT-CHAR9	0x49	0xC9
PDFFONT-CHAR10	0x4A	0xD1
PDFFONT-CHAR11	0x4B	0xD2
PDFFONT-CHAR12	0x4C	0xD3
PDFFONT-CHAR13	0x4D	0xD4
PDFFONT-CHAR14	0x4E	0xD5
PDFFONT-CHAR15	0x4F	0xD6

See the **Font Initialization Values** table later in this document for the proper values for your environment.

Example:

```

. . . . .
000450 WORKING-STORAGE SECTION.
. . . . .

000470 01 FONT-CHAR-SET          PIC X(16) .

. . . . .

000980*****
000990 PROCEDURE DIVISION.
001000*****

. . . . .

MOVE " ABCDEFGHIJKLMNO" TO FONT-CHAR-SET.
CALL 'PDFINITF' USING FONT-CHAR-SET.

. . . . .

```

7.1.2 PDFENCOD

Description:

This function will encode the data and generate a table of font characters. The user may change the values in the information record to manipulate the generated output.

Call:

```
COPY "NMPDF417.COB".

CALL 'PDFENCOD' USING PDF417-PARAMETER-INFO-REC,
                     PDF417-OUTPUT-REC,
                     IN-DATA.
```

Parameters:

```
PDF417-PARAMETER-INFO-REC  the COBOL parameter information record.
PDF417-OUTPUT-REC          the COBL output record.
IN-DATA                    the input data record to be encoded.
```

A PDF417-PARAMETER-INFO-REC record is defined as follows:

```
01 PDF417-PARAMETER-INFO-REC.
   03 PDF-VERSION              PIC 9(4)  VALUE 200.
   03 NUM-BYTES-TO-ENCODE     PIC 9(4)  VALUE 0.
   03 ASPECT-RATIO.
       05 SYMBOL-HEIGHT       PIC 9(2)  VALUE 1.
       05 SYMBOL-WIDTH        PIC 9(2)  VALUE 2.
   03 ECC-TYPE                 PIC 9(1)  VALUE 1.
   03 ECC-VALUE                PIC 9(3)  VALUE 5.
   03 NO-PAD-ECC-FLAG         PIC X(1)  VALUE 'F'.
   03 INPUT-DATA-FORMAT       PIC X(1)  VALUE 'N'.
   03 SYMBOL-TYPE              PIC X(1)  VALUE 'S'.
   03 ENCODE-METHOD         PIC X(1)  VALUE 'O'.
   03 PRINT-SYMBOL-HORIZ-ALIGN PIC X(1)  VALUE 'L'.
   03 PRINT-SYMBOL-VERT-ALIGN PIC X(1)  VALUE 'T'.
   03 EBCDIC-TO-ASCII-FLAG   PIC X(1)  VALUE 'Y'.
```

NOTES:

- Provided with the distribution is a copylib that contains these record definitions. This file is named NMPDF417 on some distributions and PDFCOPYLIB on others.
- The PDF-VERSION field *must* have a value of 200 for this version of the PDF417 encoder.
- If the fields are not set or are invalid values, then an error may be returned or the encoder's behavior may be unpredictable.
- If you are using the encoder as an AS/400 ILE service program, remember to include LINKAGE TYPE IS PROCEDURE in the call statement. Refer to the AS/400 sample programs.

A PDF417-OUTPUT-REC record is defined as follows:

```
01 PDF417-OUTPUT-REC.
   05 PRINT-SYMBOL-WIDTH      PIC 9(3)  VALUE 37.
   05 PRINT-SYMBOL-HEIGHT     PIC 9(3)  VALUE 25.
   05 RESULT-CODE             PIC 9(3)  VALUE 0.
   05 OUTPUT-LINES            PIC X(37) OCCURS 25.
```

NOTES:

- The field OUTPUT-LINES is defined as a character table. The PRINT-SYMBOL-WIDTH and PRINT-SYMBOL-HEIGHT fields must precisely reflect the size of OUTPUT-LINES, as this is the only mechanism the encode has to determine the size of this storage area.
- The PDF417 encoder does not RESULT-CODE is used at the receiver field for the status of the COBOL API. This is defined as a field in the record so as not to rely on the RETURN CODE processing under COBOL. This API does not set RETURN CODE.
- It is important to realize that the values assigned to the PRINT-SYMBOL-WIDTH and the PRINT-SYMBOL-HEIGHT are used by the encoder to determine what shape of symbol will be generated. The PRINT-SYMBOL-HEIGHT has the symbol size restriction of being within the 3-90 row range.

Example:

```

. . . . .
000450 WORKING-STORAGE SECTION.
. . . . .

01 IN-DATA.
   03 IN-DATA-TEXT                PIC X(65) VALUE
      "THIS IS A TEST OF THE SILVER BAY SOFTWARE, LLC. PDF417 E
-   "NCODER.".

*****
* PDF417 COBOL STRUCTURES USED IN API
*****
* COPY "NMPDF417.COB"

01 PDF417-PARAMETER-INFO-REC.
   03 PDF-VERSION                  PIC 9(4) VALUE 0200.
   03 NUM-BYTES-TO-ENCODE          PIC 9(4) VALUE 0.
   03 ASPECT-RATIO.
      05 SYMBOL-HEIGHT             PIC 9(2) VALUE 1.
      05 SYMBOL-WIDTH              PIC 9(2) VALUE 2.
   03 ECC-TYPE                     PIC 9(1) VALUE 1.
      88 ECC-DEFAULT                VALUE 0.
      88 ECC-LEVEL                  VALUE 1.
      88 ECC-PERCENT                 VALUE 2.
   03 ECC-VALUE                    PIC 9(3) VALUE 5.
   03 NO-PAD-ECC-FLAG              PIC X(1) VALUE 'N'.
      88 PAD-ECCLEVEL               VALUE 'Y'.
      88 NO-PAD-ECC                 VALUE 'N'.
   03 INPUT-DATA-FORMAT            PIC X(1) VALUE 'N'.
      88 NO-ECI-GLI-PROCESSING      VALUE 'N'.
      88 ECI-FORMAT                 VALUE 'E'.
      88 GLI-FORMAT                 VALUE 'G'.
      03 SYMBOL-TYPE                PIC X(1) VALUE 'S'.
      88 STANDARD-SYMBOL            VALUE 'S'.
      88 TRUNCATED-SYMBOL           VALUE 'T'.
   03 ENCODE-METHOD              PIC X(1) VALUE 'O'.
      88 ENCODE-OPTIMAL             VALUE 'O'.
      88 ENCODE-BINARY              VALUE 'B'.
      03 PRINT-SYMBOL-HORIZ-ALIGN   PIC X(1) VALUE 'L'.
      88 H-ALIGN-CENTERED           VALUE 'C'.
      88 H-ALIGN-LEFT               VALUE 'L'.

```

```

      88 H-ALIGN-RIGHT                VALUE 'R'.
03 PRINT-SYMBOL-VERT-ALIGN PIC X(1)  VALUE 'T'.
      88 V-ALIGN-CENTERED            VALUE 'C'.
      88 V-ALIGN-TOP                 VALUE 'T'.
      88 V-ALIGN-BOTTOM              VALUE 'B'.
*****
* THIS FLAG WILL NEED TO BE SET TO INDICATE IF THE DATA
* SHOULD BE CONVERTED FROM EBCDIC TO ASCII DATA PRIOR TO
* THE ENCODING PROCESS.
*****
      03 EBCDIC-TO-ASCII-FLAG PIC X(1) VALUE 'Y'.
      88 TRANSLATE                 VALUE 'Y'.
      88 NO-TRANSLATE               VALUE 'N'.

*****
* COBOL OUTPUT RECORD *
*****
* THE OUTPUT AREA WILL NEED TO BE CALCULATED BY USING THE AREA
* IN INCHES THEN CONVERTING TO PIXELS AND FINALLY CONVERTING
* TO CHARACTERS. THE API WILL CONVERT TO ROWS AND COLUMNS
* INTERNALLY AND FILL THE UNUSED AREA WITH SPACES. THE PRINT
* ROUTINE WILL PRINT THE ENTIRE AREA.
*
* E.G.:
*      WIDTH = 2.00 INCHES
*      HEIGHT = 0.75 INCHES
*
*      TO CONVERT TO PIXELS WITH 300 DPI:
*      300 * 2.0 = 600 PIXELS
*      300 * .75 = 225 PIXELS
*
* TO CONVERT TO CHARS FROM PIXELS WITH A 10 MILLAGE FONT
* DEFINITION OF 12 PIXELS PER CHARACTER WIDTH AND 9 PIXELS
* PER CHARACTER HEIGHT:
*      600 / 12 = 50 CHARACTERS IN 1 ROW
*      225 / 9 = 25 CHARACTER ROWS
*
* NOTE: ROUND DOWN THE VALUE BECAUSE THE GUARANTEE IS THAT
* THE SYMBOL WILL NOT EXCEED THIS AREA WHEN IT IS
* SET WITH FONT CHARACTERS.
*
* WIDTH = 50 CHARACTERS MAXIMUM
* HEIGHT = 25 CHARACTERS MAXIMUM
*
*****

01 PDF417-OUTPUT-REC.
      05 PRINT-SYMBOL-WIDTH PIC 9(3) VALUE 050.
      05 PRINT-SYMBOL-HEIGHT PIC 9(3) VALUE 025.
      05 RESULT-CODE PIC 9(3).
          88 PDF-OK VALUE 000.
          88 PDF-INV-PARAM VALUE 001.
          88 PDF-TOO-LONG VALUE 002.
      05 OUTPUT-LINES PIC X(50) OCCURS 25.
. . . . .

000980*****
000990 PROCEDURE DIVISION.
001000*****
. . . . .

```

```
MOVE 65 TO NUM-BYTES-TO-ENCODE.  
MOVE 0 TO RESULT-CODE.  
CALL 'PDFENCOD' USING PDF417-PARAMETER-INFO-REC,  
                        PDF417-OUTPUT-REC,  
                        IN-DATA.  
  
IF PDF-OK  
    PERFORM PRINT-SYMBOL  
ELSE  
    DISPLAY "ENCODER FAILED. "  
           " ERROR CODE = " RESULT-CODE  
    STOP RUN.
```

.

8 Using the PDF417 Encoder with Other Languages

8.1 RPG

On the AS/400 platform, RPG is a supported language. The basic API is identical to the COBOL API described in Sections 6 and 7. The sample programs that come with the encoder provide the corresponding record formats.

8.2 Other Languages

Calling the Silver Bay Software PDF417 encoder API with any language is technically feasible, however, *it is up to the user to setup the interface*. The API is written in the 'C' language. All of the parameters must be passed by reference (address), not by value. Within the distribution there is sample code written for C and COBOL. Depending on the installation platform, there is example code for other languages. Any use of the encoder with an unsupported language does *not* fall into the general product support area; however, documentation of nuances and environment specifics may be added to this document upon request.

9 Font and Printing-Related Information

Using the Silver Bay Software PDF417 encoder and the “font rendering” process, a PDF417 symbol is printed using a special font. This font consists of characters representing bars and spaces (that is, black regions and white regions). The “output” of the encoder is simply a set of characters which, when printed using our font, will result in a PDF417 symbol.

The encoder does not actually deliver these characters to the printer; it is the responsibility of you, the application programmer, to integrate these output characters in your printer output stream.

Each “row” of characters in the output buffer represents a single row of the PDF417 symbol. Each output character however represents four adjacent modules.

9.1 PDF417 Font Basics

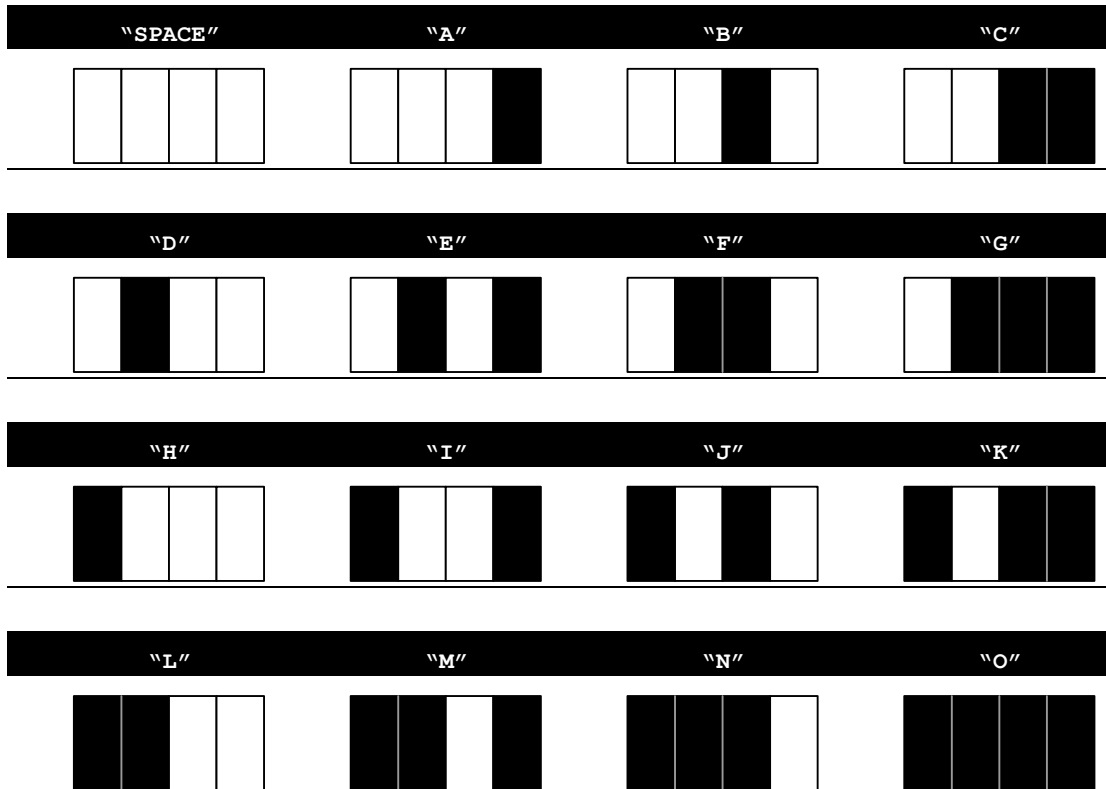
Most printers today support the use of scaleable fonts, allowing the printer to increase or decrease code points in size. This enables the printer to render a font at virtually any size. This is usually accomplished through techniques like pixel averaging and dot resolution enhancement. Unfortunately, this presents a problem for the PDF417 symbology, as it is very important that the bars and spaces be of uniform width. For this reason, the Silver Bay Software PDF417 encoder uses fixed size fonts.

At least two fonts have been provided for each supported printer type; a “normal” font and a “large” font. The normal font is the one you will most commonly use. It generates a PDF417 symbol that is closest to the AIM specification's recommended size.

The large font is provided for situations where the symbol may be susceptible to damage and maximum readability is required. The downside to using the larger font is that it increases the size of the symbol significantly (by 33% or 50%, depending on your printer density).

9.2 The Character Set

Each font character, or code point, in the PDF417 font, represents four modules of a PDF417 symbol. The characters used are diagrammed below (not to scale):



The outlines in the above diagrams indicate the module positions. Only sixteen code points are defined in the default font – all other code points are unused.

The actual character output of the encoder looks something like this (this the partial output from the sample COBOL program):

```

OOEDGEALCJBNALBHNIC LFB KCJILAOMADH
OOEDGMDCCBOKINOGJKLDGDF KOEAMOMADH
OOEDFJGLCCHEMKHILLOJHGICLKE OIOMADH
OOEDGMBOK ECAIJA OCBNDE HBKNOIOMADH
OOEDGELAKDNAAMF DLGJGMAFCJNAIOMADH
OOEDGJOB CFENAIHENKHLGNDFKJOIAOMADH
OOEDECMNBB IA LLDNNLLGMJOBILNAOMADH
OOEDGJDHCMADALKIHNOGMGHFLKNIFAOMADH
OOEDECCNCMCMADDANLNDGENLLCODNMOMADH
OOEDFHNGJAIJAHFJ LJF E FCBHLAIOMADH
OOEDGDNACACJACGM ODHDFBGECJGAIOMADH
OOEDEA OBHBOACHHFIHDNGCNIJHLOIOMADH

```

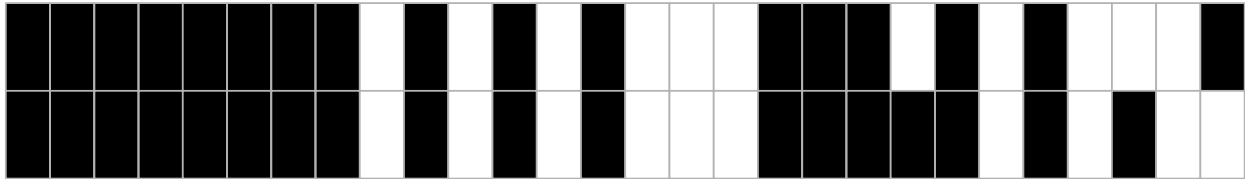
As a simple example, consider just the first few characters of the first line:

```
OOEDGEA
```

When printed using the PDF417 font, the following pattern would actually be printed (without the grey lines of course):



If we extend our example to include the first six characters of the second line of output, the resulting output (of both lines) would look like this:



You've probably noticed that every line starts with the same four characters (OOED) and also ends with the same four characters (MADH). These represent the start and stop patterns of the PDF417 symbol. Since the start and stop pattern are the same for every row in the symbol, it makes sense that the same characters would appear in every line of the output stream.

9.3 Module Size

Remember that a module is defined as the smallest black or white rectangular unit used to make up a symbol. We have designed our font such that each character in the font represents four adjacent modules. Furthermore, the height of each character is equivalent to the height of a module.

The AIM specification for PDF417 recommends that the height of each row of modules in a PDF417 symbol be three times the module width. For example, if a module width of 10 mil (10/1000ths of an inch) is being used, then the module height should be 30 mil.

For 300 dot per inch printers, a font has been provided which uses a module width of 3 printer pixels and a module height of 9 printer pixels. This is exactly 10 mil by 30 mil ($3/300 = 0.01$ and $9/300 = 0.03$), and thus very nicely satisfies the AIM recommendations.

However, for 240 dot per inch printers (many IBM AFP printers), it is not possible to generate a 10 mil PDF417 symbol (as each module would need to be 2.4 pixels wide). Thus, a font has been provided which uses a module width of 2 printer pixels and a module height of 6 printer pixels. This gives us a PDF417 symbol whose modules are 8.3 mil by 25 mil.

These dimensions are close enough to the AIM recommendations of 10 mil and 30 mil that virtually all PDF417 scanning devices will have no difficulty in reading these symbols.

9.4 Font Metrics

Silver Bay Software provides PDF417 fonts for HP PCL printers, Xerox printers, and AFP printers. Each set of fonts has different naming conventions as well as different characteristics. However, a common convention has been used for the suffix of each font.

The last 3 digits of any font's name indicates the module width and height in pixels. This is formatted as *whh* where *w* is the width and *hh* is the height. For example, the PCL font HPP3309 uses a module width of 3 pixels and a module height of 9 pixels.

9.4.1 AFP Fonts

Four fonts have been provided for AFP printers; two for 240 DPI printers and two for 300 DPI printers. Note that the 300 DPI fonts are used on 600 and 1200 DPI printers as well. The fonts are PD2206 and PD2309 for 240 DPI printers and PD3309 and PD3412 for 300 DPI printers.

The naming convention is **PDdwhh** where *d* is the printer density (2 for 240 DPI or 3 for 300 DPI), *w* is the module width in pixels, and *hh* is the module height in pixels. The actual filenames of the AFP resources are as follows:

Filename	Description
X0PD2206	240 DPI Coded Font (normal)
C0PD2206	240 DPI Character Set (normal)
X0PD2309	240 DPI Coded Font (large)
C0PD2309	240 DPI Character Set (large)
X0PD3309	300 DPI Coded Font (normal)
C0PD3309	300 DPI Character Set (normal)
X0PD3412	300 DPI Coded Font (large)
C0PD3412	300 DPI Character Set (large)
T1PDF417	Code Page (for both 240 and 300)

The following table lists line spacing information for each font:

Font Name	Lines Per Inch	Line Height	
		Pixels	Inches
PD2206	40.00	6	0.0250
PD2309	26.66	9	0.0375
PD3309	33.33	9	0.0300
PD3412	25.00	12	0.0400

The line spacing method you will use depends on your application program and your programming environment. For example, PELS or lines per inch are normally used in PAGEDEFs while AS/400 programmers using a DDS must position each line independently (and thus will use the information in the Inches column).

9.4.2 HP PCL Fonts

Four fonts have been provided for HP PCL printers. Since PCL soft fonts (downloadable) can only be 300 DPI, the same font is used on 300, 600, and 1200 dot per inch PCL printers. Both portrait and landscape fonts are included.

The naming convention is **HPPodwhh** where *o* is the font orientation (P for portrait or L for landscape), *d* is the printer density (always 3 for 300 DPI), *w* is the module width in pixels, and *hh* is the module height in pixels. The actual filenames of the HP soft fonts are as follows:

The actual filenames are as follows:

Filename	Description
HPPP3309	Portrait Font (normal)
HPPL3309	Landscape Font (normal)
HPPP3412	Portrait Font (large)
HPPL3412	Landscape Font (large)

The PDF417 PCL fonts HPP3309/HPPL3309 and HPP3412/HPL3412 print at 40 and 25 lines per inch respectively. However, PCL does not allow line spacing of these values to be set. Rather, PCL programmers must use a Vertical Motion Index; that is, for each line of the PDF417 printed the application program must position the printer's cursor to the correct vertical (and horizontal position). Vertical (and horizontal) positioning can be specified in either pixels or decipoints (720ths of an inch). The following tables lists the appropriate vertical motion information for each of the fonts:

Font Name	Pixels	Decipoints
HPPP3309	9	21.6
HPPL3309	9	21.6
HPPP3412	12	28.8
HPPL3412	12	28.8

Whether you choose to use pixel positioning (dots) or decipoint positioning is your choice. However, realize decipoint positioning has the advantage that it is printer resolution independent, thus the same PCL will work for 300, 600, and 1200 DPI printers.

9.4.3 Xerox Fonts

Twelve fonts have been provided for Xerox printers, however only six of them are valid for any given printer model. The Xerox font disk you received will contain both 9700 series and 5-word fonts. Choose and install only the appropriate fonts for your printer model.

The naming convention for these fonts is **Xfowhh** where *f* is the font series (9 for 9700 or 5 for 5-word), *o* is the font orientation (P for portrait or L for landscape), *w* is the module width in pixels, and *hh* is the module height in pixels. The actual filenames of the HP soft fonts are as follows:

The actual filenames are as follows:

Filename	Description
X5P309	5-Word Portrait Font (normal)
X5L309	5-Word Landscape Font (normal)
X5P310	5-Word Portrait Font (normal)
X5L310	5-Word Landscape Font (normal)
X5P412	5-Word Portrait Font (large)
X5L412	5-Word Landscape Font (large)
X9P309	9700 Portrait Font (normal)
X9L309	9700 Landscape Font (normal)
X9P310	9700 Portrait Font (normal)
X9L310	9700 Landscape Font (normal)
X9P412	9700 Portrait Font (large)
X9L412	9700 Landscape Font (large)

Due to the limitations of line spacing on Xerox printers, line spacing cannot be specified for the X5P309, X5L309, X9P309, and X9L309 fonts (as they require line spacing of 33.33 lines per inch). If you use these fonts, do not specify a line spacing value in the JSL.

The following table lists the line spacing using with the various Xerox fonts:

Font Name	Lines Per Inch	Inches
X5P309	33.33	0.0300
X5L309	33.33	0.0300
X5P310	30.00	0.0333
X5L310	30.00	0.0333
X5P412	25.00	0.0400
X5L412	25.00	0.0400
X9P309	33.33	0.0300
X9L309	33.33	0.0300
X9P310	30.00	0.0333
X9L310	30.00	0.0333
X9P412	25.00	0.0400
X9L412	25.00	0.0400

9.5 IBM Advanced Functional Printing (AFP)

Using AFP, font selection and line-to-line spacing is usually controlled from within the Page Definition (PAGEDEF) file. The following is a fragment from a sample PAGEDEF file for a 240 DPI printer. The sample assumes that:

- The PDF417 font being used is the 206 font and has been assigned a logical font name of "PD2206".
- The symbol is to be positioned 3.75" (900 pels) from the left edge of the page and 5" (1200 pels) from the top of the page.
- The PDF417 output has been directed to channel 2 by the application program.

```
SETUNITS 1 PELS 1 PELS LINESP 6 PELS;
PRINTLINE REPEAT 37
      POSITION 900 1200
      FONT PD2206
      CHANNEL 2;
```

Using the 2309 font would require a different vertical line spacing and a different font:

```
SETUNITS 1 PELS 1 PELS LINESP 9 PELS;
PRINTLINE REPEAT 37
      POSITION 900 1200
      FONT PD2309
      CHANNEL 2;
```

9.6 Xerox Metacode/JSL

When printing a PDF417 symbol using a Xerox printer, the PDF417 font must be installed on the printer. Transferring this font to your specific Xerox printer is beyond the scope of this document.

To print the symbol using Metacode commands, each line must be positioned as it is printed. Each of the lines of output must be printed by sending the vertical position, horizontal position, font selection, the characters comprising the line, and the terminate command. The line spacing depends on the font used; the 309 font uses a line spacing of 9 pixels, the 310 font uses 10 pixels, whereas the 412 uses 12 pixels.

Note that when using JSL it is not strictly necessary to include line spacing information. Furthermore, line spacing cannot be specified for the 309 font, as it is printing at 33.33 lines per inch (JSL only allows line spacing up to 30.00 lines per inch). By not specifying lines per inch in the JSL, the printer will use the correct line spacing information from the font itself.

9.7 Hewlett-Packard Printer Control Language (HP-PCL)

Using HP-PCL, the programmer must either install the PDF417 font(s) directly on the printer, or download the PDF417 soft font to the printer as part of the print job. The programmer must also invoke the font at the proper time. Installation of a font on a printer is beyond the scope of this document.

Downloaded soft font selection is most conveniently done by assigning the downloaded font a Font ID number, using the escape sequence $E_C^*c\#D$ (where # is the Font ID number), and then invoking the font using that number. Selecting the font may either be done by making it the

primary font, using the escape sequence $E_C(\#X$, or by making it the secondary font using the escape sequence $E_C)\#X$ and then invoking the font with the S_O character. The latter approach has the advantage that the printer can be returned to the previous primary font using the S_I character.

The following code fragment illustrates downloading the font as a soft font. This fragment would be typical of a UNIX system directly connected to a HP-PCL printer such as a Hewlett-Packard LaserJet™ family printer.

```
printf("\033*c%dD", PDF_FONT_ID); /* specify the soft font ID */
fp = fopen(fontFile, "r");
for(;;)
{
    nChars = fread(buffer, 1, sizeof(buffer), fp);
    if (nChars == 0)
        break;
    fwrite(buffer, 1, nChars, stdout);
}
fclose(fp);
printf("\033)dX", PDF_FONT_ID);          /* set the font as secondary */
```

Using HP-PCL, the simplest way to ensure the proper vertical positioning is to send a cursor positioning escape sequence before each line of output. HP-PCL provides three means of specifying position on the page – row/column, pixels, and deci-points (1/720th of an inch). The relevant escape sequences are as follows:

	Horizontal Position	Vertical Position
Row/Column	$E_C\&a\#C$	$E_C\&a\#R$
Pixels (Dots)	$E_C\&a\#X$	$E_C\&a\#Y$
Deci-points	$E_C\&a\#H$	$E_C\&a\#V$

When used with the PDF417 encoder, the row/column positioning escape sequences will not, in general, produce the desired result and so should be avoided (as they do not provide fine enough control over the position).

To use pixel positioning, send both a horizontal and vertical escape sequence before each row. Each successive line in the output should have the same horizontal position, but a vertical position that is either 9 pixels (for the 309 font) or 12 pixels (for the 412 font) farther down the page.

To use deci-point positioning, send both a horizontal and vertical escape sequences before each row. Each successive line in the output should have the same horizontal position, but a vertical position that is 21.6 decipoints farther down the page for the 309 font or 28.8 decipoints for the 412 font. Using the decipoint method has the advantage that the output is independent of the printer resolution. The following code fragment shows an example of this approach.

This example prints the symbol 1 inch from the left edge of the page, and 2 inches from the top of the page, assuming the use of the 412 font:

```
#define ASCII_SI    15    /* select primary font */
#define ASCII_SO    14    /* select secondary font */
double hPos = 720.0;    /* 1 inch */
double vPos = 1440.0/    /* 2 inches */
int row, col;
struct sPdf417Output outputData;
. . .
```

```

/* do the encode here */
. . .
putchar(ASCII_SO); /* invoke secondary (PDF417) font */
for (row = 0; row < pdfNumRows; row++)
{
    printf("\033&a%.1lfH", hPos);          /* horizontal pos */
    printf("\033&a%.1lfV", vPos + row * 28.8); /* vertical pos */
    for (col = 0; col < pdfNumCols; col++)
    {
        putchar(outputData.output[row][col]);
    }
}
putchar(ASCII_SI); /* return to primary font */

```

9.8 AS/400 DDS

As mentioned at the beginning of this section, the line spacing you will use is based on which font you select. Most DDS users are accustomed to setting line spacing using the LPI function. However, LPI cannot set “custom” line per inch settings (it can only set 4, 6, 8, 9, and 12 lines per inch). A PDF-417 symbol, even using the largest font, requires much finer line spacing.

The solution then is to use explicit POSITION calls for each line of output. The DDS must place each line of output at its precise location on the page. The table for AFP fonts provided earlier in this section lists the necessary line spacing (in thousandths of an inch). The following is a sample for a 300 DPI printer using the X0PD3309 font. Note that this font uses a line spacing of 0.0300 inches:

A			CDEFNT (X0PD3309)
A	PDFOUT01	37A	POSITION(1.700 1.350)
A	PDFOUT02	37A	POSITION(1.730 1.350)
A	PDFOUT03	37A	POSITION(1.760 1.350)
A	PDFOUT04	37A	POSITION(1.790 1.350)
A	PDFOUT05	37A	POSITION(1.820 1.350)
A	PDFOUT06	37A	POSITION(1.850 1.350)
A	PDFOUT07	37A	POSITION(1.880 1.350)
A	PDFOUT08	37A	POSITION(1.910 1.350)
A	PDFOUT09	37A	POSITION(1.940 1.350)
A	PDFOUT10	37A	POSITION(1.970 1.350)
A	PDFOUT11	37A	POSITION(2.000 1.350)
A	PDFOUT12	37A	POSITION(2.030 1.350)
A	PDFOUT13	37A	POSITION(2.060 1.350)
A	PDFOUT14	37A	POSITION(2.090 1.350)
A	PDFOUT15	37A	POSITION(2.120 1.350)
A	PDFOUT16	37A	POSITION(2.150 1.350)
A	PDFOUT17	37A	POSITION(2.180 1.350)
A	PDFOUT18	37A	POSITION(2.210 1.350)
A	PDFOUT19	37A	POSITION(2.240 1.350)
A	PDFOUT20	37A	POSITION(2.270 1.350)
A	PDFOUT21	37A	POSITION(2.300 1.350)
A	PDFOUT22	37A	POSITION(2.330 1.350)
A	PDFOUT23	37A	POSITION(2.360 1.350)
A	PDFOUT24	37A	POSITION(2.390 1.350)
A	PDFOUT25	37A	POSITION(2.420 1.350)

This DDS prints a symbol 1.70 inches down and 1.35 inches over on the page. Each of the successive 25 lines of output is 0.030 inches lower than the previous one (as each row of the PDF-417 symbol is 0.03 inches high).

10 Appendix

10.1 API Return Values

The function return value for all of the APIs is an enumerated type, PDFERROR (defined in pdfdefs.h - this file is automatically included by pdfenc.h). The table on the next page lists the error values defined for PDFERROR:

COBOL Value	C enum Value	Meaning
0	PERR_OK	Success.
1	PERR_INV_PARAM	An invalid parameter was passed. This typically indicates that a NULL pointer was passed to the encoder.
2	PERR_NO_MEMORY	Not enough memory allocated.
3	PERR_TOO_LONG	The data passed will not fit in a single PDF417 symbol.
4	PERR_INV_CODEWORD	Invalid codeword, outside of valid range (0 - 899).
5	PERR_INV_SYM_LENGTH	Invalid symbol length descriptor.
6	PERR_INV_CTRLBLCK	Invalid Macro PDF control block.
7	PERR_INV_SEGIDX	Invalid Segment Index, outside of valid range (0 - 99998).
8	PERR_INV_FILEID	Invalid file ID codeword (0 - 899).
9	PERR_INV_FLDDDES	Invalid field designator, outside of valid range (0 - 6).
10	PERR_INV_EMPTYFLD	Invalid optional field, found empty optional field.
11	PERR_INV_SHIFT_LATCH	Invalid sub-mode shift or latch.
12	PERR_INV_NC_GRPING	Invalid grouping embedded ECI within Numeric Compaction.
13	PERR_INV_BC_GRPING	Invalid grouping embedded ECI within Binary Compaction.
14	PERR_INV_ECESEQ	Invalid escape sequence in data.
15	PERR_INV_GLISEQ	Invalid GLI sequence in data.
16	PERR_INV_ECISEQ	Invalid ECI sequence in data.
17	PERR_INV_VERSION	Invalid or unsupported input structure version.
18	PERR_INV_SYMBOL	Invalid symbol.
19	PERR_INV_ROWS	Invalid number of rows, outside of valid range (3 - 90).
20	PERR_INV_COLS	Invalid number of columns, outside of valid range (1 - 30).
21	PERR_INV_ECCLEVEL	Invalid error correction level, outside valid range (0 - 9).
22	PERR_INV_ECCTYPE	Invalid ECC type choice (default, level, percent).
23	PERR_INV_ECCPERCENT	Invalid error correction percent, outside valid range (1-100).
24	PERR_INV_ECCVALUE	Invalid error correction value, outside of valid range for ECC type.
25	PERR_EXCEED_ECCLIMIT	Have exceeded the limit of the maximum ECC level.

COBOL Value	C enum Value	Meaning
26	PERR_INV_PDFTYPE	Invalid PDF417 symbol type.
27	PERR_INV_SYM_MATRIX	Invalid symbol matrix (check row or column values).
28	PERR_INV_RENDER	Invalid render info. Structure.
29	PERR_INV_SCANLINE	Invalid scanline, outside of valid scanline range.
30	PERR_SINK_FAILED	Data sink failure.
31	PERR_INV_BUFFINFO	Invalid buffer info. structure.
32	PERR_NO_DATA	No data codewords were found.
33	PERR_INV_PADFLAG	Invalid setting for pad ECC flag (TRUE or FALSE).
34	PERR_INV_DATAFORMAT	Invalid data format flag.
35	PERR_INV_ASPECT_RATIO	Invalid aspect ratio value.
36	PERR_INV_AR_MODHEIGHT	Invalid module height in aspect ratio.
37	PERR_INV_AR_MODWIDTH	Invalid module width in aspect ratio.
38	PERR_INV_AR_SYMHEIGHT	Invalid symbol height in aspect ratio.
39	PERR_INV_AR_SYMWIDTH	Invalid symbol width in aspect ratio.
40	PERR_INV_NUMBYTES	Invalid number of bytes parameter.
41	PERR_INV_USER_WRKAREA	Invalid user work area.
42	PERR_INV_OPT_WORKAREA	Invalid optimal encode work area.
43	PERR_INV_CODESET	Invalid code set value.
44	PERR_BUFFER_TOO_SMALL	Output render buffer too small.
45	PERR_INV_SHAVEWIDTH	Invalid render information – shave width
46	PERR_INV_SHAVEHEIGHT	Invalid render information – shave height
47	PERR_INV_CHARNUM	Invalid character number value (0 - 9)
48	PERR_INV_DESTBUFF	Invalid receiver for rendering a single line
49	PERR_INV_ENCMETHOD	Invalid encoding method value
50	PERR_INV_H_ALIGNMENT	Invalid horizontal alignment parameter for font rendering
51	PERR_INV_V_ALIGNMENT	Invalid vertical alignment parameter for font rendering
52	PERR_INV_CONVERTFLAG	Invalid EBCDIC to ASCII conversion flag value
53	PERR_MISMATCH_ECCTYPE	Mismatch of ECC_TYPE and ECC_VALUE (expect 0 and 0)

In addition to these values, a number of "internal" error codes are defined. All these errors have a value of 100 or greater. Should you encounter an internal error code, please note the code and contact Silver Bay Software technical support.

10.2 Symbology Technical Summary

PDF417 has the following basic characteristics:

1. Encodable character set:
 - a. Text Compaction mode permits all printable ASCII characters to be encoded, i.e. values 32 – 126 inclusive in accordance with ISO 646, as well as selected control characters.
 - b. Byte Compaction mode permits all 256 possible 8-bit byte values to be encoded. This includes all ASCII characters value 0 – 127 inclusive and provides for international character set support.
 - c. Numeric Compaction mode permits efficient encoding of long numeric data digit strings.
 - d. Up to 811,800 different character sets or data interpretations.
 - e. Various function codewords for control purposes.
2. Each codeword is represented as a PDF417 character. The character structure is: n, k, m symbology of 17 modules (n), 4 bar elements (k), with the largest element 6 modules wide (m).
3. Maximum number of data characters per symbol (at error correction level 0) 925 data codewords which can encode:
 - a. Text Compaction mode: 1850 characters (at 2 data characters per codeword).
 - b. Byte Compaction mode: 1109 characters (at 1.2 data characters per codeword).
 - c. Numeric Compaction mode: 2710 characters (at 2.93 data characters per codeword).
4. Symbol Size:
 - a. Number of rows: 3 to 90.
 - b. Number of columns: 1 to 30.
 - c. Width in modules: 90X to 583X including quiet zones.
 - d. Maximum codeword capacity: 928 codewords.
 - e. Maximum data codeword capacity: 925 codewords (other three are length descriptor and error detection codewords).
 - f. Since the number of rows and columns are selectable, the aspect ratio of a PDF417 symbol may be varied when printing to suit the spatial requirements of the application.
5. Error detection: 2 codewords per symbol.
 - a. Selectable error correction: 2 to 512 codewords per symbol.
6. Non-data overhead:
 - a. Per row: 73 modules, including quiet zones.

- b. Per symbol: 3 additional codewords, represented as symbol characters (the length descriptor and the error correction detection codewords).
7. Code type: Continuous, multi-row two-dimensional.
8. Character self-checking: Yes.
9. Bi-directionally decodeable: Yes.

The following summary is of additional features that are inherent or optional in PDF417:

1. Data Compaction (inherent): These schemes are defined to compact a number of data characters into codewords. Generally data is not directly represented on a one character for one codeword basis.
2. Extended Channel Interpretations (optional): These mechanisms allow up to 811,800 different character sets or interpretations to be encoded.
3. Macro PDF417 (optional): This mechanism allows files of data to be represented logically and consecutively in a number of PDF417 symbols. Up to 99,999 different PDF417 symbols can be linked or concatenated and be scanned in any sequence to enable the original data files to be correctly reconstructed.
4. Edge to edge decodable (inherent): As an n (modules) or k (bar elements) symbology, PDF417 can be decoded by measuring elements from edge to similar edge.
5. Partial scan 'stitching' (inherent): The combination of three characteristics in PDF417:
 - a. being synchronized horizontally, or self clocking.
 - b. row identification.
 - c. being vertically synchronized, by using cluster values to achieve local row discrimination allows a single linear scan to cross a number of rows and achieve a partial decode of the data so long as at least one complete symbol character per row is decoded into its codeword value. The decoding algorithm can then 'stitch' the individual codewords into a meaningful matrix.
6. Error correction (inherent): A user may define one of 9 error correction levels. All but Level 0 not only detect errors but also can correct erroneously decoded or missing codewords.

Compact PDF417 (optional): In relatively 'clean' environments, it is possible to reduce some of the row overhead to improve the symbol density.

NOTE: In earlier versions of PDF417, this was called Truncated PDF417. Compact PDF417 is the preferred term to avoid confusion with the more general use of the term 'truncated'.

10.3 Font Initialization Values

The following tables list the supported printer platforms and the code points to use with the encoder's font initialization function. Refer to the PdfFontInitRender (for C programming) or PDFINITF (for COBOL programming) section for more information on font initialization. Failure to have the proper code points in the program can cause unpredictable

output and can even cause printer reboots. The default font character set, using the provided printer fonts, should work for most print environments.

The information has been broken into two tables; one for UNIX and PC Platforms (ASCII platforms) and a second for IBM S/370 and AS/400 systems (EBCDIC platforms).

The code point values are provided in hexadecimal and are listed in the order required by the initialization function (SPACE, CHAR1 through CHAR15).

UNIX and PC Platform

Printer Family	Code Points (in HEX)
IBM AFP	20, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F
Xerox DJDE	20, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F
Xerox MetaCode	20, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F
HP PCL	20, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F

IBM S/370, S/390

Printer Family	Code Points (in HEX)
IBM AFP	40, C1, C2, C3, C4, C5, C6, C7, C8, C9, D1, D2, D3, D4, D5, D6
Xerox DJDE	40, C1, C2, C3, C4, C5, C6, C7, C8, C9, D1, D2, D3, D4, D5, D6
Xerox MetaCode	20, 41, 42, 43, 44, 45, 46, 47, 48, 49, 4A, 4B, 4C, 4D, 4E, 4F
HP PCL	40, C1, C2, C3, C4, C5, C6, C7, C8, C9, D1, D2, D3, D4, D5, D6