



# Adding PDF output to existing systems

---

Michael McKeough

**This paper addresses the issues involved in adding PDF as an output form to an existing system, using Adobe PDF library, and Datalogics DLI.**

---

## 1.0 Background

---

Generally, there is a continuum of complexity in adding PDF support to an existing system. At the least complex end of this continuum are systems which already support multiple output formats. These generally have an internally defined interface which all drawing operations address, which then re-addresses such operations per output format. Such an existing architecture makes it easy to add a new output format, simply translating the internal forms of command to the new format command. At the most complex end of the spectrum is an existing system that supports only a single format, and thus has interface elements to that format spread over an unpredictably large amount of code. Converting such a system to support PDF is sometimes too difficult to attempt. In the middle are systems that group some of the interface to the display, or that have multiple separate groups of interfaces to the display. Lastly, squarely in the middle, are systems that depend on a complex operating system facility (Like GDI on Windows, or QuickDraw on Macs) to supply a rich and complex layout capability. In these cases, the interface is generally fairly clear, but the capabilities to be replaced are quite complex.

In addition to the complexity of the application itself, there is the complexity of the output. There are three types of data that make up the output of a text creating engine. These are Text itself, Bit Mapped Images, and Line Drawing images. Each of these may be used in varying degrees of complexity, and may be intermixed within a given set of images. Text is characterized by its encoding, its font, its size, and its layout. Bit Mapped Images are characterized by the standard in which they are encoded, their color model, resolution, and size. Line drawings may be created as external elements, and included, or may be drawn by the engine. Graphs, Charts, etc. are generally drawn within a rendering engine. DLI provides support for each of these distinctly.

Next, there is the issue of the overall form of the rendered output. PDF as a format presumes that a rendering is broken into segments that will fit onto sheets of paper (Pages). Many rendering engines do not make that assumption themselves, but rather create a stream of image which is intended to be scrolled through on a display device, or displayed as a scroll on a printer, allowing the printer itself to break the stream into pages. Renderings which are not inherently page based will be more complex to move to PDF, since PDF must break the rendering into pages.

Finally, there is the issue of things which may be done in a PDF document, which may not be done in other formats. Interactive features, like embedded links, or sound bytes, or movies. Three dimensional images. Navigation features like BookMarks. These very often add value to a rendering, but will make the conversion more difficult, particularly when the information such additional features are based on is not already present in the rendering engine. A PDF document allows accessibility functions, such as text readers, to make a document more accessible to those with disabilities. However, for these functions to work at their best, it is useful to include in the PDF document information about the document's structure and reading order (identifying paragraphs, articles, flows, tables, providing readable captions for illustrations, and so on). This level of information is usually lost in the composition functions, unless specifically added.

## **2.0 Colors**

---

Color is one of the more interesting topics of typesetting, that is never talked about. We all experience color directly, we deal with it all the time, so we never give it a thought. Most existing applications deal with color using the Crayola method. They set up a list of color names, and allow the user to choose one. On output, they convert the name to an RGB or CMYK description of the color. For most applications, this is all that is needed or required.

Colors are used in several ways. The graphics state contains a reference to two separate color models and colors. These are the *fill* color, and the *stroke* color. The fill color is used for any operation that fills an area. The stroke color is used for any operation that draws over a line in color. Additionally, every bit-map image has a color model associated with it, directly or indirectly. For most, this model is a part of the image itself, and determines the meaning of the individual color samples. In the case of a *mask* bitmap, the fill color in effect at the time the image is drawn become the image color.

### **2.1 “Device” Colors**

The Adobe PDF Library (APDFL) describes colors in a number of different ways. At the bottom, there are three device color models (Gray, RGB, and CMYK). These describe colors as an intensity between 0 and 1 of a colorant or colorants. For Gray, there is one colorant, and it is “White”. For RGB, there are three colorants, and they are Red, Green, and Blue. For CMYK, there are 4 (Cyan, Magenta, Yellow, and Black). The colors described by these models are not particularly standardized. That is, if you specify an RGB color of [0, 1, 0], you will get something pretty much greenish, but it can vary from green to green.

### **2.2 “Calibrated” Colors**

The second level of color models is Calibrated colors. These include CalGray, CalRGB, LAB, and ICCBased. The first three specify color in terms of illumination, using the CIE 1931, and, in essence, are based on the architecture of the human eye to express color in a device independent manner. The last, ICCBased, is based on the standards promulgated by the International Color Consortium to promote the use and adoption of open, vendor-neutral, cross-platform color management systems. In practice, these are used identically to the Device colors, but the results when rendered should be reliably repeatable across a very wide range of display devices. The ICCBased color profile permits specification of up to 15 colorants. In practice, it usually specifies 1, 3, or 4. The use of ICCBased colors is becoming very common, and helps to maintain color integrity over a range of rendering devices.

### **2.3 “Special” Colors**

APDFL also defines a number of Special color models. These are: Separation, DeviceN, Indexed, and Patterned. The first three of these are color models like unto the Device and Calibrated colors; the last is used to accomplish various special effects in filling areas.

### **2.3.1 Separation Colors**

A Separation color space is used to define a single colorant. It is defined by name, rather than appearance. The name may be anything, but is most often something like a Pantone color name. Since it is not possible for any rendering device to understand this name, we must also define an alternate color space, and appearance for this color. These colors are generally used in pages that are to be printed via press, and that will use the named colorant as a selector of the ink. Generally, they are created by making a separation of the page by colors.

### **2.3.2 DeviceN Colors**

A DeviceN Color space is both an extension of the concept of the Separation color space, and an extension of the concept of a process color spaces. A DeviceN color space may have any number of colorants. For instance, if a document is to be printed in only two colors, then those two colors can be expressed as Separation Colors, and a DeviceN color space can merge them into a single color. Some applications might also wish to use a 6 color process color (such as the PANTONE Hexachrome system) to express color. DeviceN allows the application designer the freedom to accomplish this.

### **2.3.3 Indexed Colors**

An Indexed color space is used to create a “palette” of colors. It uses one of the other colors spaces (any except pattern or index) to define an array of “N” specific colors, then uses only an index of from 0 to N-1 to select a color. These are used primarily within Images, where the image contains the color table. However, they may be used anywhere in APDFL.

### **2.3.4 Patterned Colors**

Pattern color spaces are used for filling areas only, never for stroking lines. There are two basic types of patterns. One uses an image, either a bitmap or a line drawing (or combination of these) repeated over the area to be filled: this is a tiled pattern. The other draws a gradient of color across the area to be filled: this is a shading pattern.

In general, only more advanced composition engines will ever use either of these. The tiled fill is often used to fill areas in a generated bar chart. I have seldom seen the shaded patterns used.

#### **2.3.4.1 Tiling Patterns**

The tile itself is essentially a PDF page. It is generally a very small page, but it has all of the qualities of a page (a content stream and a resource dictionary). This pattern is described as a rectangle of a given size. We further describe how far apart each repetition is to be, horizontally and vertically. When filling an area, the pattern is started at the current transformation matrix [0, 0], and repeated horizontally and vertically until the area to be filled is covered. Any repetitions outside of the area being filled are clipped (not drawn).

The content of the cell may determine its color, or the color may be the selected using any color model other than pattern, at the time the fill color is set. There are many restrictions on the design of the cell for the latter to be used.

#### **2.3.4.2 Shading Patterns**

A shading pattern defines a gradient of color across an area. There are 7 different types of patterns available in PDF, from a simple function which defines the color at each point in a domain, to the gradients normally used to express shading across 3 dimensional free form artifacts.

### **2.4 Opacity**

In most display devices, ink is considered to be opaque. That is, if I paint an area of the page RED, then a bit later, paint the same area BLUE, the result will be that the area is BLUE. This works well for display screens and simple printers, but it is false in fact for the general case of printing. In fact, if I print RED over an area, and later print BLUE over it, I am likely to have an area of MAGENTA. The actual color that appears depends on the Opacity, or Translucency, of the ink used.

In the last few years, the model of ink in APDFL has been modified to allow for translucency. The usage and control of Translucent ink is itself a very large subject. Its control is in the graphic state, through several parameters which determine an “alpha” or translucency value for any given drawing operation.

### **3.0 Text**

---

Text appears, on the surface, to be easy. It can be the hardest of the marking operators to accomplish correctly though. A great many factors come into play when adding text to a document. It must be noted that neither DLI nor APDFL is intended to be a text composition language. It is presumed by both that text composition is done by the application.

#### **3.1 Encodings**

Within a computer system, text is always encoded. Often though, we assume an encoding, without ever really thinking about it. When we want to set type in one script, and one language, we can get away with that. When we are required to use multiple scripts and languages, it becomes a great deal harder.

Most of the systems used in the West presume that text is encoded in a one or another variant of ISO-Latin ([ISO 8859](#)). These are character sets of 256 characters each, standardized in the mid 1980s. Systems which use these character sets generally just did so, with no mention what character set was in use, but being careful to align the fonts and printers to match the input character set. Additionally, many nations established encoding schemas for their own languages and writing scripts. Collectively, these encodings may be referred to as National Encoding Schemas.

Starting in 1991, Unicode was assembled. This is a single encoding which encompasses all of the know scripts and languages in the world. A unicode character is a 24 bit numeric code. There are compression schemas which allow the characters to be carried principally as 32 bits, 16 bits, or 8 bits (UTF32, UTF16, and UTF8). A more complete description of Unicode can be seen at the [Unicode Consortium's Home Page](#).

DLI contains operators which allow any national encoding to be translated to Unicode, and any Unicode string to be translated to a national encoding. It also allows fonts to be created which use Unicode as their character encoding, as well as fonts which use National Encodings as their character encoding. This facility is provided for DLI via "International Components for Unicode" ([ICU](#)).

The choice of character encoding affects more than just the appearance of a document. The ability to search for text within a document, and recognize textual constructs in the input to a document, is often conditioned by the choice of encoding. In a PDF document, one can permit searching of text in any form through the use of encoding conversion tables embedded in the document. DLI always creates and embeds such tables when possible.

#### **3.2 Fonts**

There are several types of fonts that may be used in a PDF document, and several ways any given font may be used. PDF documents may use Adobe Type 1 fonts, Adobe Type 3 fonts (Though I have never seen anyone use one), Adobe Type 0 CID fonts, TrueType Fonts, and OpenType Fonts. Most of these use a single byte per character (with a maximum characters per font of 256). Two forms, Type 0 CID (Adobe) and Type 2 CID (TrueType), use a 16 bit character identifier, allowing for 64,000 characters per font. The 16 bit character fonts may be coupled with an Open Type type 10 cmap (as a part of

the font), to allow direct use of UTF-16 encoded Unicode (including characters outside of the BMP).

A font in PDF may be included by reference (for instance, as “use Courier”), or it may be embedded in the PDF document in whole, or in part (subset). In general, including a font by reference is done only when we know the reader of the document; know that he has the desired fonts available; and know that the document is expected to have a very short lifetime (for instance, a Statement sent to a customer via the web may assume the presence of fonts, and omit embedding fonts to lower the file size). In most cases, we want to embed all fonts, and subset all embedded fonts, so that only the minimum number of characters are present. This produces a document that can assuredly be displayed in archival time frames.

DLI has operators to create fonts, both embedded and referenced, Subset and Not Subset, using all of the font formats. At font creation time, the user is required to specify the Encoding to be used for a font, and a given PDF font may use only a single Encoding. DLI will automatically minimize font object creation (i.e., if the same font is used many times, DLI will create and share a single font object). Additionally, DLI will permit the user to define fonts, based on TrueType font technology only, which are to be used for setting Unicode type. These fonts are created differently than other fonts, and permit the Unicode code points to be directly used in the PDF document. Support for operations unique to Unicode is provided for such fonts, via ICU. These include operations such as Arabic Shaping, Bi-directional text, and character composition.

A font object is, by definition, unique to a document. DLI, however, maintains a copy of all fonts used in a “cache” document, so that they may be re-used in second and subsequent documents much more quickly. In systems which create many PDF documents in a single pass, this becomes a considerable savings.

### **3.3 Appearance**

In addition to the shape and encoding provided by the font, Type appearance is affected by a number of other attributes. These are controlled via the two data structures PDE-GraphicState, and PDETextState. The graphic state structure controls the color of strokes and fills (separately) used in drawing text. This control includes opacity and translucency controls for text drawing as well. The text state structure specifies that space is to be inserted or removed between each character, or between a word space character and the following character; ratio of character width to character height; space between adjacent lines of text (in the event that operators which set line relative one another are used); baseline movement for text which is displaced from the baseline (for instance, superscript and subscript); and rendering mode.

Text in PDF may be rendered as a single solid color, filling each character (the most usual way, rendering mode 0); or as an outline of each character (rendering mode 1); or as both a fill and an outline (using the two different colors specified in Graphic State for fill color, and stroke color) (rendering mode 2). We may also insert text into a document, and have it be invisible (neither stroked nor filled) using rendering mode 3. This is sometimes used when we want text to be searchable, but not visible. Rendering modes 4 through 7 are identical to 0 through 3, except that the characters are added to the current clipping path, in addition to being painted, such that a subsequent drawing operation may fill the characters with a pattern.

Frequently, other appearance attributes are attributed to text. These include features like underlining, kerning, small caps, and inferior or superior text which may be considered attributes of type in many composition engines. In APDFL, these are NOT functions of type definition, but rather operations which may be accomplished, separately from type style.

### **3.4 Blocks of type**

APDFL is not intended to be a composition engine. That is, it is not intended to allow the user to supply a string of text, and have APDFL break it into lines of even length, positioned and justified in the space provided. APDFL presumes that the division of text into lines has already been done: each line provided to it should include an X/Y coordinate from the left edge of the baseline to be placed.

DLI and APDFL do contain facilities to aid an application in formatting text. These allow the measurement of the width (or height for vertical type) of a string of text, the calculation of the bounding box that would enclose a text string, and the ability to set and remove text.

APDFL does contain the facility to “mark” blocks of text to indicate the flow of reading order through and among pages. There is currently no support for such marking specifically present in DLI, although such marking can be done at the lowest level within DLI. There are facilities to create such marks in the PDE level of APDFL.

## **4.0 Line Drawings**

---

Line drawings are used to create images which are redrawn to the target resolution at rendering time. APDFL has a sparse language for drawing images, which is sufficient to draw any image. DLI has a somewhat richer language, which encompasses both APDFL, PostScript (PS), and some frequently used operators outside of both sets. What DLI does in these cases is to translate a higher level construct (such as an Arc) into the APDFL primitive operations (Splines).

### **4.1 Path Drawing Operators**

The basis of all line drawing is the ability to describe a path, then either stroke (draw a mark along) that path, or fill (flood the area enclosed by the path) it. Paths may be arbitrarily complex, and the colors used to draw them may vary from simple basic color primitives (Black/White or Red/Green/Blue etc.) through colors with alpha channels, to colors specified with ICC profiles, including translucency. Colors may also specify a pattern to be used in drawing, or gradients.

APDFL itself contains line drawing operators to draw a line, a spline, or a rectangle. These actually are sufficient to draw any object, but most composition engines which support PostScript use a larger set. For this reason, DLI also supports arcs, both circular and elliptical, and PieSlice. DLI allows these to be added to a path, or drawn directly to content.

### **4.2 Forms XObjects**

APDFL permits the re-use of marking operators. This allows a designer to specify that some structure of marks is to be defined separately, and used repeatedly. In APDFL, this construct is a Forms XObject. The greatest value of this is that it allows for the reduction of content in a document. For instance, if I am printing 100,000 pages of a statement, I can move all of the constant content of the statement into one or two Forms XObjects, and put only the variable data into individual pages. This can result in a very great savings not only in file size, but also in transfer time, and, with the proper rendering agent, in rendering time.

DLI has a very simple to use interface for creating Forms XObjects, and for placing them into pages. They may be scaled and rotated at usage time as well as at creation time. However, to be a useful facility, the application has to be able to recognize constructs that are constant, and segregate them into separate constructs in the page. This is usually where the difficulty arises in existing applications.

### **4.3 PDF As an Image Format**

A PDF page may be converted into a Forms XObject, and inserted into a page as an Image. This has been a difficult thing to accomplish in APDFL directly (prior to Release. 7.0), but can be done with a single call in DLI. The page used as an image may itself contain other XObject resources, as well as Font and Image resources. These are imported into the using document along with the page data. One problem that can occur when doing this is if the imported page contains subset fonts. These fonts will generally NOT be merged into other subsets of the same font, resulting in a larger than desired PDF file.

## 5.0 Bit Map Images

---

Most pictures included in documents are in the form of bit maps. These are collections of Picture Elements, arrayed into rows and columns. They are defined in terms of the model of color represented in each picture element, the width of a row, and the height of a column.

Some images also contain a second image, as a “mask” to place over the image as drawn, allowing what is under the image to show through. Some images accomplish this same thing by using a color or channel to carry “alpha” information about the image.

PDF allows such images to be used only if they are converted to a form compatible with PDF: in essence, a stream whose content is the picture elements, and whose dictionary describes the picture. These dictionaries may be stored as resources (Image XObjects), or used directly in the PDF content (In-line images).

DLI contains convertors to allow a number of common image formats to be used as PDF Image XObjects. DLI may be asked to convert an image file encoded in JPEG, PNG, TIFF, GIF, or BMP into a PDF image. It will also convert a PDF page to a PDF Forms XObject, which may then be treated as an image. The specified image may be contained in a file system, or in a block of memory known to the application.

DLI will automatically cache and reuse image conversions within the limits of a init/term of DLI. That is, if an image is used 100 times, the application does not need to be concerned if that particular image was used before. DLI will minimize the processing needed for that image.

Any image for which a bitmap is present may be converted by DLI into a form usable by PDF. In addition to the bitmap, the user must know the color model, width in pixels, depth in pixels, and line and row progression orders.

Many of the image formats do not carry a specific size, as a geometric limit, for the image. That is, we can know that they are 72 pixels wide, but we don't know if they are intended to be 1 inch, or 1/10th of an inch wide. Where the image resolution is known, DLI will set a Bounding Box which shows that size. When it is not known, it is presumed to be 72 DPI.

## **6.0 PDF Extensions**

---

PDF allows the user to accomplish effects that cannot be displayed on paper, and that most other display forms do not permit.

### **6.1 Bookmarks**

Bookmarks are a navigation aid. In essence, these form a table of contents that can be viewed at the same time as the pages themselves. The collection of bookmarks form a tree, where each entry links to a given page, or a given point or area on a page, and may also contain sub entries.

Most often, bookmarks are used as a single table of contents. In a highly structured document, you may want to consider making book mark trees in several different orders. For instance, you can make a tree of all illustrations, a tree of all tables, a tree of all links, as well as a tree of chapters, heads, and subheads. All of these trees then become entries in a higher level tree. The entries in the book mark tree are NOT limited to document reading order, nor are they limited to one reference to one point.

DLI contains a mechanism for easily defining bookmark tree collections.

### **6.2 Links**

A link is a subset of annotations available in DLI. A link is used to guide the reader from one place in a document, to another place in the same, or a different, document. PDF allows a very great range of links, both in terms of their targets (where the link too), and the appearance to be applied to the target (open it in the same or another window, scale the target are to fill the window, or align the top or bottom of the target to the window, and so on). When used within or between documents, the target may be a page and an area, defined in the linking document, or it may be a name, defined as a page and area in the linked-to document.

When creating a document, if you know that you will be linking other documents to it, it is a good idea to define names of link targets within the document. In this way, even if the target document is replaced, the links into it will remain valid (so long as the replacement document uses the same names).

### **6.3 Thumbnails**

Most PDF viewers offer a thumbnail as a navigation device. This is a very small representation of the page. Current viewers are all able to generate thumbnails on demand, so that it is not necessary to generate and store thumbnails in the document. However, pages can be viewed a bit more quickly if thumbnails are present.

### **6.4 Signatures**

Digital Signatures allow for assurance that a document originated from a given signer, and has not been changed since it was signed. This is accomplished use Digests of the content of the document, encrypted with the users identity, confirmed by a Signature Certificate. The signed document will be considered valid ONLY if the regeneration of the Digest matches the original Digest, after decrypting using the signature.

## **7.0 Memory**

---

DLI/APDFL use significant amounts of memory. The actual content of a document is maintained in a disc based storage cache, by default, but DLI maintains a small reference for each page until the document is completed. In cases where large documents are created, leaving the cache on a disc is often the best choice. However, when we are creating many small documents, one may trade off memory for disc access, and move the cache into memory.

Normally, PDF documents are written directly to a disc. Again, if the application wishes to manipulate files after writing (such as combining many PDF files into a single file, or storing PDF in a database, or transmitting the newly created PDF to a remote location), then the file may be created in a memory buffer.

Normally, images are read from a file on a disc. However, when images are coming from a remote location, or from a database, it may be more desirable to store them in memory, and read them directly from there. DLI permits this.

## 8.0 Considerations for a new application

---

If you are creating a new application that wishes to use PDF as an output medium, then you need to be aware that pages created in PDF can be displayed to any Printer on a Windows or Macintosh platform, and any PostScript printer on a Unix platform. They can also be rendered to bitmaps, or displayed in Windows, on any platform. So, depending on your application, you may not need to use any other output than DLI/APDFL to create a range of output displays. This can considerably simplify application development.

APDFL contains support for locating all of the fonts available via the OS for all platforms, and so can form the basis for a font selection mechanism. DLI/APDFL is fully compatible with UCS, and so can support internationalized applications.

If you are able to design applications that can retain input structure information (chapters, sections, paragraphs, tables/cells/rows/columns), this information can be incorporated into the PDF structure, creating more readily-accessible documents. Such documents can also be converted back to source documents more easily, increasing the life cycle of the document contents.

Copyright (c) 2007, Datalogics, Inc.

Datalogics, the Datalogics Logo and all Datalogics product names are either trademarks or registered trademarks of Datalogics, Inc. All other trademarks are the property of their respective owners. Reproduction of this document in whole or in part without the express written consent of Datalogics, Inc. is prohibited.

This document and related materials and information are provided “as is” with no warranties, express or implied, including but not limited to any implied warranty of merchantability, fitness for a particular purpose, non-infringement of intellectual property rights, or any warranty otherwise arising out of any proposal, specification, or sample. Datalogics, Inc. assumes no responsibility for any errors contained in this document and has no liabilities or obligations for any damages arising from or in connection with the use of this document.