Technical Note TN1171

LaserWriter 8.6: How to Write a Converter Plug-in for the Download Manager

CONTENTS

This document describes in detail how to write a plug-in converter module for use with the Download Manager under `PrintingLib` version 8.6 (included with LaserWriter 8 version 8.6) and later. It contains information about the pieces a plug-in must have, discussion of a sample plug-in, and tips for plug-in developers.

Note:
Note: This document refers to Download Manager plug-ins as low-level converters to avoid confusion with other types of plug-ins available on the Mac OS. This terminology matches that used in other Technotes related to the Download Manager and its plug-ins.

This Technote is directed at application developers who wish to write plug-in converters.

Updated: [Jun 21 1999]

Overview

A low-level converter is used by clients of the Download Manager to convert a file or stream of a given data type (or types) into PostScript output. For example, in Mac OS 8.5, the desktop printing software is a Download Manager client that offers drag and drop printing of files to the targeted desktop printer (DTP). When the target is a PostScript printer and a low-level converter is available to handle the conversion, that low-level converter can be used to generate the PostScript code to be sent to the device, without requiring a separate application. The Download Manager and its low-level converters are described in more detail in Technote 1169, "Download Manager."

Printing plug-in files reside in the "Printing Plug-ins" folder in the Extensions folder. Each plug-in file can contain multiple plug-in libraries. Each plug-in file contains a `'PLGN'` resource indicating what shared libraries are contained in that file and what plug-in type each library is. The `PrintingLib` file itself contains many plug-ins, including several

low-level converters for the Download Manager.

> **Note:**
> `PrintingLib` is special in that the Download Manager locates plug-ins within `PrintingLib`, even though it is not in the "Printing Plug-ins" folder.

Figure 1 below gives an overview of the Download Manager's relationship to its clients and the low-level converters.

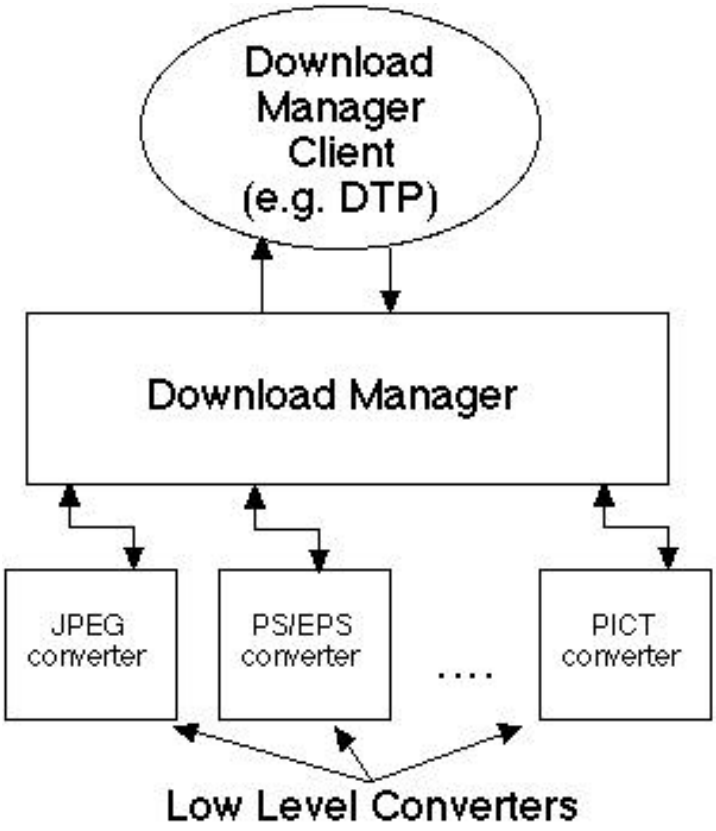

Figure 1

Back to top

Requirements

There are several requirements for a plug-in to work with the Download Manager:

1. To be seen as a Download Manager low-level converter, a plug-in file must contain a `'PLGN'` resource with ID `-8192`. This resource contains the information which identifies what type of plug-ins are contained inside the file as well as the name of the shared library containing a given plug-in. Details on this resource are documented in the `'PLGN'` Resource section of this Technote.
2. For a given plug-in shared library to be a low-level converter for use with the Download Manager, it must export a minimum set of required routines. The Download Manager calls these routines to determine whether a given low-level converter can handle a specific data type and, if so, to call the plug-in to perform the conversion of the data into PostScript output.
3. In addition to the required exported routines, a low-level converter must provide a data structure to advertise the types of data that the plug-in can handle. When asked whether it can download a given file or stream of data, the Download Manager uses this information to reduce the set of possible converters to only those which might be able to handle the data. It then asks each of those converters directly if it can handle the data and, if there is one, uses the best converter

found to proceed with the download.

Back to top

The 'PLGN' Resource

The Download Manager uses the Printing Plug-ins Manager to manage its plug-ins. For a Download Manager converter to be seen by the Printing Plug-ins Manager, it must have a resource of type `'PLGN'` with ID number `-8192`. If the plug-in does not contain this resource, it cannot be used and is ignored. Plug-ins are also required to have a standard `'cfrg'` resource describing the code fragments in the data fork of the file. For developers interested in using the Printing Plug-in Manager, more information is available in Technote 1170: "The Printing Plug-ins Manager."

The `'PLGN'` resource contains information about how many shared libraries are contained in this file and for each shared library, the type of plug-in that it is, the subtype that the library handles and the library name. To be used by the Download Manager, a plug-in must have a type of `'down'` and a subtype of `'????'`. There are no constraints on the library name beyond those imposed by the Code Fragment Manager.

The `'PLGN'` resource is defined as follows (using Rez syntax):

```
type 'PLGN' {
 integer = $$Countof(PluginLibInfo);
 array PluginLibInfo {
  literal longint; /* Type */
  literal longint; /* subtype */
  pstring;   /* library name */
  align word;
 };
};
```

A ResEdit template resource (`'TMPL'`) for the `'PLGN'` resource is contained within `PrintingLib` versions 8.6 and later.

The `PluginLibInfo` structure in C syntax is:

```
typedef OSType SettingsDataType;
typedef OSType SettingsDataSubType;

short num; // the number of shared libraries this 'PLGN' describes
PluginLibInfo libInfo[num];

typedef struct PluginLibInfo{
 SettingsDataType type;
 SettingsDataSubType subtype;
 unsigned char libraryName[];  // pascal string
        // word aligned
}PluginLibInfo;
```

- `type` is the type of plug-in that is described by this `PluginLibInfo`
- `subtype` is the subtype of data that can be handled by the plug-in described by this `PluginLibInfo`
- `libraryName` is the library name of the code fragment in the plug-in file described by this `PluginLibInfo`

Note:
A single file can contain multiple plug-in libraries.

Back to top

Required Routines

This section describes each of the routines required by the Download Manager. All of the routines described here are discussed in more detail in Technote 1169: "Download Manager." The descriptions here are intended to provide just an overview.

This discussion of the routines and the order in which a converter should call them is tailored to the way the Download Manager calls a converter in response to the Finder's desktop printing software. In addition, this discussion assumes that the Download Manager client is downloading a file. This document discusses the differences between downloading files and downloading data from other sources in the section Input Stream Types.

psLowGetConverterInformation

Before it can determine which converters can handle a given download job, the Download Manager must determine the capabilities of all the available converters. It does this by calling the `psLowGetConverterInformation` routine for each low-level converter. This routine returns a pointer to a `ConverterDescription` structure which provides a list of data types that a given converter can handle. The `ConverterDescription` structure contains additional information which can be used to narrow the search further. A sample `ConverterDescription` structure is described in detail later in this document in the section Sample Converter Description.

Note:
The Download Manager caches the `ConverterDescription` structures it obtains from each low-level converter to improve performance. This is not an issue for users, but during development of a low-level converter it does require a developer to do some special handling of the plug-in files. See the Tips section near the end of the document for more information.

psLowCanConvert

When the desktop printing software asks the Download Manager if it can handle a given file, the Download Manager checks the `ConverterDescription` structures for all of the low-level converters to obtain a list of candidate converters. For each low-level converter on its list of candidates, the Download Manager calls the converter's `psLowCanConvert` routine. This allows the converter an opportunity to examine the data to determine whether it can indeed be handled by the converter and if so, with what "priority" it can handle the data.

Each converter returns a "priority", an indication of how well it can handle the given type of data. It is possible for multiple converters to handle data of a given type. If there are multiple low-level converters which can handle the data, the converter which returns the highest priority is chosen to convert the data.

The data to convert is provided to the low-level converter as a `PSStream` structure which contains routines to allow reading and writing of the data. The sample code demonstrates use of the `PSStream` structures. There is also some additional discussion of the `PSStream` structure and the routines to read and write to `PSStreams` in Appendix A.

psLowGetStreamInfo

If the data can be handled by a low-level converter, the Download Manager tells the desktop printing software that it can download the file. At that point, the desktop printing software wants more information about the type of data that it is downloading. Since the file is an opaque object to both the desktop printing software requesting the download and to the Download Manager itself, the Download Manager calls the converter's `psLowGetStreamInfo` routine to obtain more information about the file, such as the number of pages, the type of data, the number of copies that are being generated, whether the download job is manual feed, and so forth. This information is used to provide information to the user about the Download Manager print jobs in a given DTP queue.

psLowPeekConvert

Once a file to be downloaded reaches the top of a DTP queue, the desktop printing software asks the Download Manager to download the file. In response, the Download Manager first calls the low-level converter's `psLowPeekConvert` routine. This gives the converter an opportunity to look at the data and record any information that might be useful when it converts the data into PostScript output. For example, the low-level converter built into `PrintingLib` which handles downloading of PostScript and EPS data first parses the PostScript Document Structuring Conventions (DSC) comments in the PostScript file, and records what fonts are required by the document. This allows the converter to request appropriate printer queries and to incorporate the query information during the generation of the PostScript output

psLowAddConverterQueries

After calling the `psLowPeekConvert` routine, the Download Manager calls the low-level converter's `psLowAddConverterQueries` routine. This allows a low-level converter to tell the Download Manager what queries it desires. The Download Manager is responsible for performing the queries. The sample code demonstrates use of some of the queries, and Appendix B has more information about the available queries.

psLowDoConvert

After the Download Manager performs the queries, it calls the low-level converter's `psLowDoConvert` routine to do the conversion. At this point, the low-level converter is required to convert the input into PostScript output. The Download Manager itself generates no PostScript output as part of its operation; it relies on the low-level converter to do this. It is the responsibility of the low-level converter to generate all the PostScript output for the download job, including any or all printer feature requests. Support for handling feature code generation is available through the `FeatureUtilsLib` library built into `PrintingLib`; however, it is up to a given low-level converter to make the appropriate calls if it chooses to utilize this library. The sample code demonstrates use of the `FeatureUtilsLib` library. Appendix C has more information about `FeatureUtilsLib`.

The data is provided to the low-level converter as an input `PSStream` structure which contains a routine to read the data. The generated PostScript output is written to an output `PSStream` structure which contains a routine to write the data to the output device. The sample code demonstrates use of the PSStream structures. There is also additional discussion of the routines to read and write to `PSStreams` in Appendix A.

In addition to generating all the PostScript output and writing it to the output stream, it is the responsibility of the low-level converter to read data from the output stream and write it back to the input stream. Data read from the output stream consists of data returned from a PostScript output device. This can be error information or other types of status information. Writing such data back to the input stream allows the Download Manager to process this information appropriately.

psLowGetConverterVersion

One final routine must be supplied by a low-level converter to provide version information. The routine `psLowGetConverterVersion` allows a caller to determine CFM version information for a given low-level converter.

Back to top

Other Details

This section discusses some low-level converter issues in additional detail. The sample low-level converter code addresses each issue in depth, and there is sample code to support the discussion points.

The ConverterDescription Structure

The low-level converter routine psLowGetConverterInformation returns a pointer to a ConverterDescription structure. The purpose of the ConverterDescription structure is to advertise the types of data that a low-level converter can handle. The ConverterDescription structure is loosely modeled after the DriverDescription structure used for PCI Drivers. The ConverterDescription structure is defined as follows:

```
typedef struct ConverterDescription {
 OSType converterDescSignature;
 ConverterDescVersion converterDescVersion;
 ConverterType converterType;
 ConverterService converterService;
}ConverterDescription;
```

The converterDescSignature field in the ConverterDescription structure is required to be a signature long word designating this to be a converter description structure. The value of this signature is:

```
 enum {
 kTheConverterDescriptionSignature = 'dhwu'
  /*first long word of ConverterDescription*/
};
```

The converterDescVersion field (long word) of the ConverterDescription structure indicates the version of the structure being used. This is used to distinguish different versions of converter descriptions which have the same signature but different values. This is defined as follows:

```
typedef UInt32 ConverterDescVersion;
enum {
 kInitialConverterDescriptor = 0
   /* the initial version of ConverterDescription
    supported by the Download Manager
   */
};
```

The next field of the ConverterDescription is the converterType. This structure contains name and information string data as well as the converter module version information. It is defined as:

```
typedef struct ConverterType{
 Str31  name;
 Str255 info;
 NumVersion version;
}ConverterType;

typedef struct NumVersion{
 UInt8 majorRev; /*1st part of version number in BCD*/
 UInt8 minorAndBugRev; /*2nd and 3rd part of version
      number share a byte*/
 UInt8 stage; /*stage code: dev, alpha, beta, final*/
 UInt8 nonRelRev; /*rev level of nonreleased version*/
}NumVersion;
```

The final field in the `ConverterDescription` structure is a `ConverterService` structure which contains information about what types of data the converter can handle. This is defined as:

```
typedef struct ConverterService{
 UInt32 nTypes;
 ConverterTypeInfo typeInfo[1];
}ConverterService;

typedef struct ConverterTypeInfo{
 OSType type;
 Fixed priority;
 Str15 matchString;
}ConverterTypeInfo;
```

A converter may be able to handle files or streams of different data types. The `nTypes` field is the number of different `ConverterTypeInfo` structures contained in the `ConverterService`.

The type field of the `ConverterTypeInfo` structure is the `OSType` of data described by the `ConverterTypeInfo`. If the converter can handle any type, it should include the type `'****'` (i.e., the wildcard type) with the appropriate `matchString`.

The `matchString` field is a Pascal string of at most 15 bytes (plus a length byte) corresponding to any identification bytes the converter requires at the beginning of the data. For example, a PostScript converter requires the identification data '%!' to be the first 2 bytes of data. A converter informs the Download Manager what types of data it can convert by supplying the data type and the `matchString`. For a given converter, if there is no unique `matchString` for the `OSType` of the `ConverterTypeInfo`, the length of the string should be set to 0. This indicates to the Download Manager that this `OSType` does not have a magic identification string. An example of this kind of converter is the PICT converter, since the first 512 bytes of a PICT data file can contain any data.

The Download Manager uses the `ConverterTypeInfo` data to determine the list of low-level converters which can possibly be used to download the data. It does this by looking at the first 15 bytes of data and uses the `ConverterTypeInfo` data to determine which low-level converters may support the data. After paring down the list with this information, it normally calls the `psLowCanConvert` routine of each of the possible low-level converters to allow further examination of the data.

In some cases the Download Manager cannot call the psLowCanConvert routine of the candidate low-level converters. This is the case where the data is supplied from a PSStream which cannot be repositioned or randomly accessed since reading the data in psLowCanConvert would prevent the data from being downloaded later. In these cases, the Download Manager uses the priority field of the ConverterTypeInfo data to determine whether the low-level converter can handle the data. See the section Input Stream Types for more information about the handling of stream types which cannot be randomly accessed.

The priority field in a ConverterTypeInfo structure is a Fixed number which is the priority estimate of the converter for handling the type of data described by the type field and the matchString. This priority is used by the Download Manager when *only* the matchString and type of the data being downloaded are available for determining whether a converter can handle the download. In all other cases, the Download Manager calls the psLowCanConvert function with a stream that the low-level converter can use to determine whether it can handle the data. For this reason, the priority specified here should be the priority that the converter can guarantee based *only* on the OSType and the matchString data. If a matchString of 0 is provided, the priority should probably be 0x0 (i.e., cannot convert without looking at more data). If the converter cannot handle a stream which cannot be randomly accessed, it should assign a priority of 0x0 for that OSType in the ConverterTypeInfo.

Note:
A given low-level converter may have more than one ConverterTypeInfo for a given type. This would occur if there was more than one priority and matchString pair appropriate for a given data type. An example would be a converter which can handle both GIF87a and GIF89a. These files have the same type, but would have different matchStrings.

A sample ConverterDescription structure is part of the sample cde discussed later in the Sample Converter Description section.

Back to top

Input Stream Types

The data to be converted by a low-level converter is provided via a PSStream structure. The PSStream structure is a union of a number of different types of streams.

There are currently two types of PSStream structures which can be provided to low-level converters as input streams:

1. The type of PSStream used when downloading files is of type kPSRandomAccessStream. This type of stream represents data that can be accessed randomly, i.e., the position where the next read from the stream occurs can be changed. All low-level converters must be able to read data from this type of stream.
2. The other type of PSStream that low-level converters might see is kPSSerialStream. This type of stream does not have the ability to position the next read; instead, the data is only available in a sequential fashion. This type of stream will not be seen when converting files with desktop printing, but may instead be seen when the Download Manager is called by other clients. For data generated by a Download Manager client on the fly, there may be no way to position the read mark within the data stream.

The psLowCanConvert routine is not called by the Download Manager for streams which cannot be rewound since there would be no point in doing so. See the ConverterDescription discussion above regarding the Download Manager selection of low-level converters in this case. In addition, the Download Manager does not call a low-level converter's psLowPeekConvert routine if the input stream is of type kPSSerialStream since such a stream can, by definition, only be read once.

A given low-level converter should be able to operate with either type of stream. When processing data types that don't require random access, this should be fairly straightforward. Ideally, peeking at the data is not required and will only improve the quality of the PostScript output.

Writing the PostScript Language Output

During execution of its psLowDoConvert routine, a low-level converter writes its PostScript output to the output stream. The simplest way to do this is to make a call to the write procedure on the output stream with the data to be written. The major disadvantage of this approach is that the Download Manager client (such as the desktop printing software) gets no detailed status information about the progress of the conversion. For example, there is no information about what page is currently being printed since only the low-level converter has this information.

To allow for the communication of status information about the data being written to a stream, the PSStream structure for the types kPSRandomAccessStream and kPSSerialStream contains a PSPosition structure which allows a low-level converter to tag the data it is writing with additional information. This tag information is loosely designed around the PostScript Document Structuring Conventions. The intent is that low-level converters tag the portions of the PostScript output which correspond to the various DSC comments. This allows the Download Manager and its clients to track the progress of the download and other information about the PostScript output.

The library PSUtilsLib in PrintingLib contains routines that may be useful to low-level converters. Some of these routines are helpful for generating the tagged output. In addition, PSUtilsLib contains routines which are useful for generating formatted output (similar to printf in the standard C library).

Appendix A has more information about the PSStream and PSPosition structures, as well as the routines in the PSUtilsLib library which can be used for generating formatted output and tagging that output.

Reading the Back Channel

During the conversion process in the psLowDoConvert routine, a low-level converter is expected to read data from the output stream and write that data back to the input stream. This allows the Download Manager and its clients to detect any PostScript errors or status messages that come back from the output device. The conversion process of the psLowDoConvert routine resembles Figure 2 below:

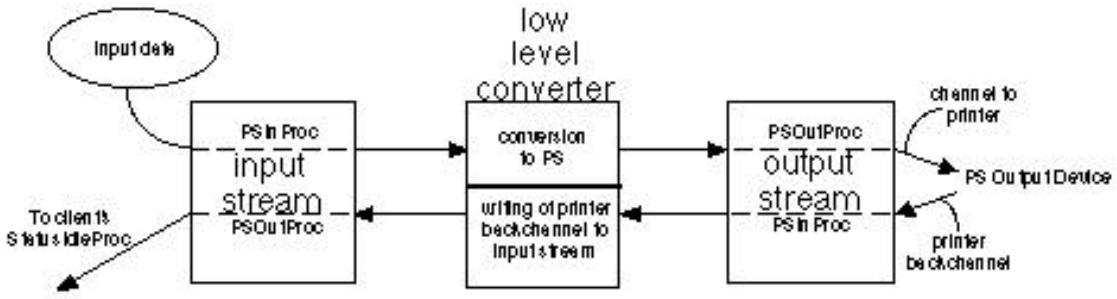

Figure 2

Private Data Hints

Most of the low-level converter routines have a Collection parameter passed to them. The purpose of passing a Collection to a low-level converter routine is to pass hints about the job requirements and to provide a container for arbitrary data to or from the routine's caller.

A low-level converter which gathers information during its peek phase (psLowPeekConvert) may want that information to be available during the conversion phase (psLowDoConvert) of the download. The only reliable way to pass data from one routine to the other is through the Collection which is passed into both routines. Note that global data is not appropriate to use for this purpose, since the converter module may be unloaded between the calls to psLowPeekConvert and psLowDoConvert.

The sample code described later in this document demonstrates passing collections to low-level routines. It also demonstrates the appropriate way of using private data hints to pass data from the peek phase to the conversion phase.

Warning and Error Logging

A low-level converter operates without an explicit user interface and should not display any dialogs or alerts to the user. To communicate warning or error conditions, the low-level converter should use the PSPosition portion of the PSStream structure used for the conversion.

In particular, the PSSubsections of kSubLogErrorData and kSubLogWarningData are used to tag messages as error or warning messages, respectively. Warnings are non-fatal conditions that might be used to alert a user. Errors are considered fatal. After a warning is generated, the converter should proceed normally. If an error is generated, the converter should terminate its conversion immediately after writing the tagged error message.

More information about the use of these subsections is contained in Technote 1169: "The Download Manager." Additionally the sample code uses kSubLogErrorData as needed.

When a low-level converter reports these conditions, the Download Manager passes the information to its client (such as the desktop printing software). In addition, the Download Manager has the ability to log this data to a log file. Normally this feature is disabled, but it can be enabled by a sophisticated user or developer. See the Tips portion of this document to see how to enable logging and how it might be useful during testing and development.

Back to top

Sample Code Overview

Most of the remainder of this document discusses a sample low-level converter which converts JPEG/JFIF data into PostScript output suitable for transmission to PostScript Level 2 and PostScript 3 output devices. The sample code is structured in a way that is intended to make it straightforward for developers to modify it to support conversion of graphics formats that are output as a single page. Examples include PNG and GIF. The sample code should also be relatively straightforward to modify to support multiple-page documents.

The discussion about the sample code is divided into a number of sections. The section Sample JPEG Converter Specification provides a high level discussion of the sample code features. This is intended to provide an overview of the goals of this sample low-level converter. Within that specification is information about implementation. This information does not contain anything about the structure of the sample code, but is simply to provide an overview of the implementation.

The section Introduction to the Sample Code Structure discusses the sample code's structure in some detail. In particular, it discusses the way the sample code is broken out into a "shell" which provides the support for printer feature handling, for the DSC structure of the PostScript output, and for the tagging of the output so that the Download Manager and its clients can report status. This "shell" code is usable as is for low-level converters other than a JPEG converter, particularly those that generate 1 page of output.

The section JPEG Converter Specific Code discusses the portion of the sample code which has been tailored for handling JPEG output. This portion of the sample code relies on the 'shell' code to call it appropriately. This code is the guts of what makes this particular low-level converter a "JPEG" low-level converter.

The low-level converter Shell Code section describes the "shell" code in some detail. This discussion is for those who want to understand everything about the sample code and/or for those who wish to create a low-level converter that handles more than a single page of output.

Note:
You should download <link> the sample code before continuing with this section.

Back to top

Sample JPEG Converter Specification

This portion of the document describes the sample JPEG converter module for use with the Download Manager. This module is a so-called 'low-level converter,' which simply means that it is a plug-in module that the Download Manager can call to 'convert' a stream of data of a certain type into a PostScript output stream. In the case of the sample JPEG converter module described here, conversion of the data into a PostScript stream means transformation of the raw JPEG or JFIF data into a stream more suited to the target PostScript output device.Some details are:

- PostScript output devices supporting PostScript Language Level 2 or higher can be sent the JPEG data as is,

without performing any image compression. This sample code does not support generation of output suitable for a PostScript Level 1 output device.

- The converter centers the JPEG image on the page and orients the image so that the longest edge of the image is parallel to the longest edge of the paper. This means that images whose width is greater than the height are printed in landscape orientation. In addition, if the image is larger than the imageable area of the page, the image is uniformly scaled so that it fits in the imageable area of the page. If the image is smaller than the page in both dimensions, no scaling adjustment is done.
- There is support for desktop printers which are "Save to File" printers. If the target output device advertises itself as requiring an EPS stream to be generated, the JPEG converter generates EPS data into the output stream. There is no screen preview generated, but the data is EPS, including the bounding box.
- For non-EPS output, most print time features from the Print dialog are invoked. The feature settings are chosen from the saved defaults for the target output device.

One significant goal of the JPEG converter module is that the output it produces conforms to the DSC. The converter module generates the necessary and appropriate DSC comments into the output stream.

Overall Strategy

The basic strategy of the sample JPEG converter module is to determine first if the data stream is JPEG data that it can handle. The JPEG converter can handle raw JPEG data as well as JFIF data. The only known case of valid JPEG data that the converter cannot handle is "progressive JPEG," an extension to the original JPEG specification. Progressive JPEG data cannot be handled by a PostScript Level 2 or PostScript 3 output device directly. If the data is either invalid JPEG data or is "progressive JPEG" data, the JPEG converter reports that it cannot handle the data.

Since this sample code cannot handle printing to Level 1 printers, the sample code checks the target language level and, if it is Level 1 or includes Level 1 (such as "Save as Level 1 Compatible"), it reports it cannot handle the file.

When the JPEG converter can handle the data, it merely adds DSC header comments and a small PostScript "wrapper" around the JPEG data itself. For target output requiring support for ASCII data, the JPEG data is transformed into ASCII85 data on the host. In this case, the decompressed data is wrapped with a slightly different PostScript wrapper to ensure that it prints correctly.

Print Dialog Feature Support

The JPEG converter attempts to support most of the print dialog features normally associated with a standard print job. Since the user does not normally get a print dialog to select print time features when using the Download Manager or the JPEG converter module, the default features for the target desktop printer or output device are used. That is, the user gets the same result as if they had brought up the print dialog and clicked **Print** without adjusting any settings in the print dialog.

> **Note:**
> These statements assume that the client invoking the original download has used the Download Manager routine `psCreateDMJobCollection` to create the hints collection passed to the Download Manager downloading routines. This is true for drag and drop desktop printing in Mac OS 8.5 and later.

The saved defaults for these features are used as the print time values by the JPEG converter:

- Number of copies.
- Paper source.
- Cover Page Handling.
- Duplex (if available).
- Error Handling: PostScript and Tray Switching.
- Save to File defaults, including whether to default to save to file.
- Any PPD features available for the target output device.
- Save to disk DTP support.

Note:
If the user has not saved defaults for this desktop printer, the standard print dialog defaults apply for non-printer specific features (1 copy, no cover page, no PostScript error handling, and so forth). In this case, printer specific features are treated as "Printer's Default", no PostScript code is sent to activate those features and the current printer configuration is used. Finally, the paper handling is treated as Automatic Feed for the default paper size as specified in the PPD file.

When printing to a "Save to Disk" DTP or to a printer which has "Save to File" as its default, a disk file is written instead of output being sent to the printer. The JPEG converter configures itself as described by the user's saved defaults for the Print dialog save panel. The user's choice of "PostScript Job" or EPS is respected; although no EPS previews are created. In addition, the `LanguageLevel`, ASCII/Binary selection and font inclusion defaults are specified (of course, JPEG files have no fonts, but other converters may need to include fonts).

Note:
There is currently no support for "Save as PDF". Currently, if "Save as PDF" is the default, the Download Manager requests PostScript Job instead.

Some features from the Print Dialog are ignored. These features are:

- Print Time: foreground/background makes little sense and users requiring a special print time handling must use the desktop printing software to set it.
- Calibrated Color settings.
- N-up Printing. Any settings are ignored so the default values of 1 sheet per page with no border are always used, regardless of the user's saved defaults.

One sticky point is the issue of paper size. For normal print jobs through the Printing Manager, the paper size is based on the print record, which in turn is based on the Page Setup dialog choices and the default print record. Since the user has no way of choosing these through the Download Manager, the default paper size is the default as given in the `*DefaultPageSize` keyword in the PPD file assigned to the target output device.

One additional feature is that any `*DeviceAdjustMatrix` or `*PatchCode` entries in the target PPD file are utilized properly.

Implementation Details

This section describes how the JPEG converter, as a plug-in for the Download Manager, implements each of its required routines.

psLowGetConverterInformation

The JPEG converter returns a pointer to a `ConverterDescription` structure which reflects its capabilities. The name of the JPEG Converter is "Sample JPEG Downloader." The info string for the converter is obtained from a `'STR#'` resource with id value `JPEGCONVERTERSTRINGS_ID` and the string number `kJPEGConverterInfoString`. This resource is defined in sample LangEnglish.r.

The `ConverterDescription` for the JPEG converter reports that the JPEG converter can handle three types of data: `'JPEG'`, `'JFIF'` and `'????'`. For each of these data types, the `ConverterDescription` for the JPEG converter requires the first two bytes of the file to be `0xFFD8`. The priority for each of these types in the `ConverterDescription` is zero, meaning that the JPEG converter cannot handle the data unless it can read more than the first 15 bytes of data to determine if it is valid JPEG data. This is a conservative approach since without reading the data, the sample converter can't be sure that it has valid JPEG data, or whether it is of a format (progressive JPEG) that it can not handle.

psLowCanConvert

This routine is required to determine if the JPEG converter can handle the data stream and if so, with what priority. To determine this, the JPEG converter first determines if the output device requires Level 1 support. If it does require Level

1 support or has unknown PostScript support, the JPEG converter reports that it cannot convert the file.

> **Note:**
> The language level information available at the time `psLowCanConvert` is called may be more conservative than the true output `LanguageLevel`. That is, if the user has not set up her printer, the `LanguageLevel` is considered unknown. If this is the case, this sample converter cannot support the data, even though the printer may support Level 2 PostScript. Use a `LanguageLevel` 2 printer and set up that printer in the Chooser to test this sample converter.

If the PostScript `LanguageLevel` of the output device is `LanguageLevel` 2 or greater, the JPEG converter looks at the stream of data and determines if it is valid JPEG data that it can handle. If so, it obtains the width, height, and depth of the JPEG data. If the data is valid JPEG data that it can handle, i.e., the width and height are non-zero and the depth is either 8 bit or 24 bit, it returns a priority of 10. If not, it returns a priority of 0, meaning that it cannot handle the data.

psLowGetStreamInfo

This routine is used by a client to get some minimal information about the downloaded data type, and so forth. The JPEG converter reports the following:

- Download data type: `'JFIF'`.
- Number of pages: 1.
- Manual feed is determined by calling the routine `psRequiresManualFeed` in `FeatureUtilsLib` for non-EPS stream creation. When creating an EPS output stream, manual feed is always false.
- Number of copies: When creating a non-EPS output stream, the number of copies is that requested as the print time default. When creating an EPS output stream, the number of copies is 1.

These are the only pieces of information that the JPEG converter reports. Any other pieces of information requested are reported as Unknown.

psLowPeekConvert

The JPEG converter does not need to peek at the data, so its `psLowPeekConvert` routine merely returns without peeking.

psLowAddConverterQueries

The JPEG converter adds queries for the PostScript `LanguageLevel` and the channel characteristics.

The PostScript `LanguageLevel` query is used, at conversion time, to determine what the PostScript `LanguageLevel` really is. This allows the converter to verify at print time whether it can really support the target output device. The only situations where this could fail after the converter checked the language level in `psLowCanConvert` are when:

- the caller requesting the download changed the target output device between its calls to `psCanDownloadFile` and `psDownloadFile` and the new device requires Level 1 support
- the target output device changed between the time of printer setup and the time of the print job and the new device requires Level 1 support
- The queries about the channel are used to determine whether the JPEG converter must generate binary or ASCII data.

The query hints used for these queries are:

- For `LanguageLevel`: `kHintLanguageLevelTag`, `kHintLanguageLevelId`, default Unknown.
- For channel characteristics: `kHintEighthBitTag`, `kHintEighthBitId`, default 7bit; `kHintTransparentChannelTag`, `kHintTransparentChannelId`, default notTransparent.

psLowDoConvert

The purpose of the `psLowDoConvert` call is to generate the PostScript data to the output device. During this process, the

converter reads the input stream and writes it to the output stream. In addition, it reads the output stream for data being echoed from the output device and writes such data to the input stream. During the reading and writing of the PostScript data, the JPEG converter module tailors its output stream to the target output device and provides status information to the client which has called it.

The JPEG converter first allocates its buffers for reading data from the input stream and for reading data from any back channel that might exist. It then reads its first buffer of data from the stream and verifies that the data is valid JPEG data. It does this to obtain the width, height, and depth of the stream it is to convert. If, for any reason, the data cannot be handled, the converter reports this by writing a log message `kJPEGBadDataMsg`. This is discussed in detail in the section Generating Error Messages. If the data cannot be handled, the converter returns the error `errCantHandleThisDownloadData`.

The converter then evaluates the query results. If the query for `LanguageLevel` indicates Level 1 compatible output is required, the JPEG converter cannot download the data and returns the error `errCantHandleThisDownloadData`. This should rarely happen.

The query for the channel allows the converter to configure whether it can write binary output.

If the converter can handle the data but the output stream requires ASCII data, the compressed JPEG data is encoded with ASCII85 encoding on the host before being written to the output stream. Note that in this case the converter uses the `ASCII85Decode` filter in combination with the `DCTDecode` filter in the PostScript wrapper. Once the JPEG data has been written to the output stream, the finishing PostScript wrapper is written.

Note that if the source JPEG data is one component, a grayscale image is produced regardless of the color capabilities of the target device.

Generating Error Messages

The JPEG converter module generates error messages when it detects problems with the conversion. These error messages are in addition to any provided by the Download Manager itself, such as PostScript errors or other error conditions in the output device. The error messages initiated by the JPEG converter are those relating to its ability to convert the JPEG data into PostScript output for the current output device. They are provided to the downloading client and ultimately may be provided to the user in a useful form.

There are two cases where error messages are generated. The first occurs if, after a call to `psLowCanConvert` determines that the JPEG data can be handled by the JPEG converter, but during `psLowDoConvert`, the JPEG converter determines that the JPEG data is not properly formed, it generates an error message corresponding to `kJPEGBadDataMsg`. Currently this error message, found in sample LangEnglish.r, is "The image data depth or size cannot be supported by the sample JPEG converter." This condition should not occur, but the JPEG converter is prepared to handle it if it does.

The second case where the sample JPEG converter initiates an error message is if during `psLowDoConvert`, it determines that output compatible with a PostScript `LanguageLevel` 1 device is required, it generates the error message corresponding to `JPEGNoLevel1SupportMsg`. Currently this error message, found in sample LangEnglish.r, is "Generating Level 1 Compatible PostScript output with the sample JPEG converter is not possible.". This should happen rarely, if at all. It should only happen if the user does a Chooser setup and the target output device reports to be `LanguageLevel` 2 and later the user changes the actual target output device to one that only supports `LanguageLevel` 1 output.

Back to top

Introduction to the Sample Code Structure

The sample JPEG converter is structured in a manner that is intended to make it easy for developers to create a new low-level converter to convert data formats that generate a single page of graphics. Even if a given data format generates more than a single page, the structure of the sample code is likely to ease the creation of new low-level converters.

The structure of the sample code consists of two parts. The first part is a "shell" portion which provides the support for printer feature handling, for the DSC structure of the PostScript language output, and for the tagging of the output so that the Download Manager and its clients can report status. This "shell" code is not specific to any data type and hopefully has

very few assumptions about what the output of a given low-level converter should be. The "shell" code consists of the files sample ConverterShell.c, sample ConverterShell.h, Utilities.c and Utilities.h. This code is described in detail in the section low-level converter "Shell" Code.

The shell code is specifically written to make the creation of one-page graphic converters especially easy but does contain a significant amount of code that would make more extensive converters straightforward to write. The existing structure expects that there is only one page, and it is hard coded as such. The code is commented to indicate where the single page assumptions are and developers are free to use this code as a basis for making a converter that handles more than one page.

The second part of the sample code is the part which is specific to the sample JPEG converter. It is what makes this low-level converter a JPEG converter as compared to a converter for GIF, FAX data, or other types of data. The files which comprise this portion of the code are sample JPEGConverterLib.c, sample JPEGConverterLib.h, sample JPEGConverterLib.r, and sample LangEnglish.r. The details of this portion of the implementation are described in the section JPEG Converter Specific Code.

The remainder of this section provides basic information about what the shell code does and what a user of the shell code needs to provide to use the shell code without modification.

Shell Basics

The shell code supplies all the routines that need to be exported by a low-level converter for use by the Download Manager. This should make it easier to build a low-level converter that meets all the requirements of the Download Manager. The routines exported by the shell code call special routines which are supplied by the non-shell code. The term "shell client code" will be used to refer to this non-shell portion of a low-level converter that uses the shell code.

The shell code handles a set of basic queries and gives the shell client code an opportunity to add additional queries should they be required. The basic queries it handles include:

- PostScript language level.
- ASCII/Binary.
- Color output device.
- Whether the output device is configured to generate color separations.

In addition to specifying these queries, the shell code handles the results of the query to create information in a form useful to shell client code.

For example, the color output device query and color separations query are used to compute the Boolean `canDoGrayOnHost` that is passed to the shell client code. This Boolean lets the shell client know whether it is safe to downsample any color data on the host to grayscale data. For some output types, this would allow a substantial performance benefit. For example, the JPEG converter built into `PrintingLib` knows how to handle PostScript Level 1 output devices and, when generating Level 1 compatible output and `canDoGrayOnHost` is true, it downsamples the uncompressed JPEG data on the host to reduce the amount of transmitted data by 2/3.

Shell client code doesn't have to worry about generating any feature code, cover pages, or document level DSC comments: these are handled by the shell code. Since the shell code also handles the initial portions of the page level DSC comments, shell client code does not need to generate any DSC comments other than those necessary for the PostScript stream to draw a given page.

The shell code creates and uses a `StreamInfoData` data structure. It also passes this structure to the client shell routines that it calls to emit PostScript code into the output stream. The `StreamInfoData` structure contains information about the output stream and its characteristics, such as whether it supports the low 32 characters in the 7-bit data range (transparent) and whether it supports data in the 8-bit character range. This structure is the structure which is passed to many of the `PSUtilsLib` output routines that are available for use to write data into the PostScript output stream. Some of the routines in `PSUtilsLib` can write formatted output (similar to `printf` in the standard C library) and they take into account the channel characteristics when they generate output. For more information on `PSUtilsLib` see Appendix A.

Before calling the shell client code to draw the actual page, the shell code performs scaling of the PostScript coordinate system so that the bounding box of the graphic is centered on the page and scaled to fit on the page, if appropriate. This generally gives attractive results for 1-page graphics formats that might have not fit on a single page or that would have been clipped by the imageable area. This 'auto scaling' may not be appropriate for some data types or graphics formats. This request for auto scaling may be removed from the shell code if a given low-level converter does not want this functionality.

> **Note:**
> The auto scaling PostScript is not emitted if the Download Manager job requests an EPS job or if the Download Manager job contains the `kHintDoAutoScalingTag` hint with the value false.

Details on how the shell code performs its duties are described in the section low-level converter "Shell" Code.

> **Crucial Note:**
> The sample code builds a library which is marked as using shared global data. What this means is that no matter how many simultaneous users of a given plug-in there are, they all share the same global data. For this sample converter, that is perfectly reasonable since it has no global data that maintains its current state; having shared global data saves memory.

Using shared global data does have at least one side effect that is reflected in the source code. In particular, accessing the resource fork of the plug-in file must be done carefully. The approach taken in the sample code is to access the resource fork by opening and closing it each time the converter needs to access data from the resource fork. Another approach is to open it at the beginning of the relevant routine and close it before ending that routine. An approach which does not work with shared global data is to open the resource fork in the library fragment initialization routine and expect that resource fork to be available to all clients. This does not work because the resource fork is only in the resource chain of the application which first loads the plug-in. Other uses of the plug-in after it has been loaded do not call the library fragment initialization routine if the library is marked with shared global data. Attempts to use the resource fork in this situation fail.

Shell Usage

The .c file called "sample ConverterShell.c" contains the shell code. This file is normally not modified by a user of the shell code. It consists of the exported routines needed by the Download Manager. It implements those exported routines by doing as much as it can in a generic fashion, while calling specific routines to be implemented by a low-level converter.

The routines to be implemented by a low-level converter that uses the shell code and which are called by the shell code are:

- converterGetConverterInfoPtr
- converterCanConvert
- converterGetConverterDocType
- converterPeekConvert
- converterGetVersion
- converterAddAdditionalQueries
- converterGetConverterName
- converterInitDoConvertClientData
- converterDisposeDoConvertClientData
- converterGetBBox
- converterEmitProlog
- converterEmitPageData

Here are the descriptions of these routines called by the shell code:

converterGetConverterInfoPtr

```
OSStatus converterGetConverterInfoPtr(const ConverterDescription
        **thePtr);
```

This routine is used by the shell code during psLowGetConverterInformation to obtain the `ConverterDescriptionPtr` to pass back to the Download Manager. A converter should store its `ConverterDescriptionPtr` in `*thePtr`.

converterCanConvert

```
OSStatus converterCanConvert(PSSerialStream *readStream,
 PSStream *inputStream, Collection hints,
 LowConverterInfo *dataInfo, Fixed *downloadability);
```

This routine is used by the shell code during `psLowCanConvert` to ask a low-level converter whether it can convert the data represented by `inputStream`. For convenience, the shell code passes the following parameters:

- `readStream` is a pointer to a `PSSerialStream` from which a client can read the data.
- `inputStream` is a pointer to a `PSStream` corresponding to the input data. This form of the input stream is needed if a client needs to rewind or position the stream. Note that not all streams can be repositioned so a client must first look at the stream type of inputStream if it needs to position the stream.
- `hints` is the collection provided to supply information about the features requested for the download.
- `dataInfo` is information about the Finder type associated with the stream data.
- `downloadability` is a pointer to a `Fixed` number. The converter is expected to indicate its ability to download the data. A return value of 0 reports that the converter cannot download the data. The larger the number, the more suitable the converter is to download the data. A value of 10.0 (`Fixed`) is the largest value returned by the converters within `PrintingLib`.

converterGetConverterDocType

```
OSStatus converterGetConverterDocType(PSSerialStream *readStream,
 PSStream *inputStream, Collection hints, OSType *theType);
```

This routine is called by the shell code as part of `psLowGetStreamInfo` to obtain the 'type' of document to which the data corresponds. This routine is only called after a converter has indicated that it can handle the data.

The stream and hints information is passed to allow a converter to examine the data, should it need to read the data stream to determine the 'type' of data. A converter that only handles one type of data should not read the data, but simply return the supported type in `*theType` parameter.

converterPeekConvert

```
OSStatus converterPeekConvert(PSSerialStream *readStream,
 PSStream *inputStream, Collection hints);
```

This routine is called by the shell code during the `psLowPeekConvert` call to allow a converter to peek at the input stream and record any hints about the data which would be useful during conversion.

- `readStream` is a pointer to a `PSSerialStream` from which a client can read the data.
- `inputStream` is a pointer to a `PSStream` corresponding to the input data. This form of the input stream is needed if a client needs to rewind or position the stream. Note that not all streams can be repositioned so a client must first look at the stream type of `inputStream` if it needs to position the stream.

- hints is the collection provided to supply information about the features requested for the download and allow the low-level converter to record hints obtained during the peek phase.

converterGetVersion

```
OSStatus converterGetVersion(struct CFMVersion *version);
```

Called by the shell code as part of psLowGetConverterVersion to allow a client to determine the CFM version data of the low-level converter.

converterAddAdditionalQueries

```
OSStatus converterAddAdditionalQueries(Collection hints,
   Collection query);
```

Called by the shell code during psLowAddConverterQueries to allow a low-level converter to add additional query hints. The shell code always adds the queries:

- kHintLanguageLevelTag: the PostScript language level.
- kHintEighthBitTag: whether the channel is 8 bit clean.
- kHintTransparentChannelTag: whether the channel is transparent to the low 32 characters.
- kHintColorDeviceTag: whether the output device is a color device.
- kHintColorSepTag: whether the output device is configured to do color separations.

If a converter wants to add additional query hints, it should add them to the query collection parameter.

converterGetConverterName

```
OSStatus converterGetConverterName(Str255 converterName);
```

This routine is called by the shell code during psLowDoConvert to obtain the name of the converter. This allows the shell code to use the proper "application name" for the DSC comments relating to the document creator. It also allows any generated cover page to correctly reflect the converter's name.

```
converterInitDoConvertClientData
OSStatus converterInitDoConvertClientData(void **clientData,
 PSSerialStream *readStream, PSStream *inputStream,
 PSStream *outputStream, Collection hints,
 unsigned char *backChannelDataBuffer,
 SInt32 backChannelDataBufferSize,
 UInt32 *LanguageLevel, Boolean doBinary,
 Boolean canDoGrayOnHost, Boolean isNotEPS);
```

- `converterInitDoConvertClientData` is called during the `psLowDoConvert` phase of conversion to allow a converter to create and configure any client data that it needs for the conversion process. This routine is called before any PostScript data is generated by the shell code.
- `clientData` is a pointer to a (void *) that can be supplied by the client. `clientData` is passed to each of the other routines called by the shell code during the `psLowDoConvert` phase of the conversion.
- `readStream` is a pointer to a `PSSerialStream` from which a client can read the data.
- `inputStream` is a pointer to a `PSStream` corresponding to the input data. This form of the input stream is needed if a client needs to rewind or position the stream. Note that not all streams can be repositioned so a client must first look at the stream type of inputStream if it needs to position the stream.
- `outputStream` is a pointer to a `PSStream` to which a client should write the generated PostScript language data.
- `hints` is a collection provided to supply information about the features requested for the download and allow the low-level converter to obtain hints recorded during the peek phase.
- `backChannelDataBuffer` is a buffer (or `NULL`) allocated by the shell code for use by the `converterEmitPageData` routine to read the back channel data into. Typically, this buffer is not read directly by a converter but is instead passed to `ReadWriteBackChannel` as described in the section Shell Utility Routines.
- `backChannelDataBufferSize` is the size of the `backChannelDataBuffer` data buffer. Typically this value is not used directly by a converter, but is instead passed to `ReadWriteBackChannel` as described in the section Shell Utility Routines.
- `LanguageLevel` is a pointer to a `UInt32` indicating the target language level for output. The converter supplied routine should put the minimum `LanguageLevel` required to support the generated output. This lets the shell code generate the proper %%`LanguageLevel`: DSC comment.
- `doBinary` is a Boolean indicating whether the destination can accept binary data.
- `canDoGrayOnHost` is a Boolean indicating whether the converter can downsample grayscale output from color input. This is only true if it is known that the output device is not color capable and is not generating color separations.
- `isNotEPS` is a Boolean indicating whether the generated output should be EPS output. If the generated output is not supposed to be EPS, this value is true. If the output is supposed to be EPS, this value is false.

Shell client code should not write to the output stream during the call to `converterInitDoConvertClientData`. This routine should only be used to configure the shell client code appropriately.

converterDisposeDoConvertClientData

```
OSStatus converterDisposeDoConvertClientData(void *clientData);
```

This routine is called by the shell code during `psLowDoConvert` to allow the low-level converter to dispose of any `clientData` it allocated during `converterInitDoConvertClientData`.

converterGetBBox

```
OSStatus converterGetBBox(kHintEPSBBoxVar *bbox,
    void *clientData);
```

This routine is called by the shell code during psLowDoConvert to obtain the appropriate bounding box information for the data being downloaded.

converterEmitProlog

```
OSStatus converterEmitProlog(StreamInfoData comm,
 void *clientData);
```

This routine is called by the shell code during psLowDoConvert to allow the low-level converter to emit its prolog code into the output stream. The shell code generates the appropriate %%BeginProlog and %%EndProlog comments around the prolog code emitted by converterEmitProlog.

- comm is a StreamInfoData corresponding to the output stream.
- clientData is the client data filled in by the converter when converterInitDoConvertClientData was called.

converterEmitPageData

```
OSStatus converterEmitPageData(StreamInfoData comm,
 void *clientData);
```

This routine is called by the shell code during psLowDoConvert to allow the low-level converter to emit the PostScript code to render the page into the output stream. This is where the bulk of the PostScript code emitted by the low-level converter should be generated. The shell code generates the showpage command as well as the appropriate comments after the page and job.

The PostScript coordinate system in force at the time this call is made has been set by the shell code to be the default PostScript coordinate system as modified by any device adjust matrix and any autoscaling necessary to center and scale the bounding box reported by converterGetBBox. A low-level converter should emit PostScript code to render the page so that it has a bounding box as reported by converterGetBBox.

- comm is a StreamInfoData corresponding to the output stream.
- clientData is the client data filled in by the converter when converterInitDoConvertClientData was called.

Resources

Once these routines have been implemented, the C code is complete, but the converter is not. A converter module must also have the appropriate resources, in particular the 'PLGN' resource it requires as a Download Manager plug-in. The sample code sample JPEGConverterLib.r file contains a 'PLGN' resource with ID number -8192. A converter MUST contain this resource and the libraryName field of the PluginLibInfo resource must have the name of the converter module library in place of that for the sample.

```
resource 'PLGN' (-8192,
#if qNames
 "Plug-In Info",
#endif
 purgeable) {
 {
  'down', '????', "YourConverterModuleLibName"
 }
};
```

Note:
Without a proper `'PLGN'` resource, a low-level converter will not be recognized by the Download Manager.

The name "YourConverterModuleLibName" above should be replaced with the name used for the converter code fragment.

That should be it. Once the converter module is built correctly (get that library name to match in both the `'cfrg'` resource and the `'PLGN'` resource!), you can drop an alias of that library into your "Printing Plug-ins" folder in your Extensions folder. You should now be able to test drag and drop printing in the Finder in Mac OS 8.5 or later using your plug-in.

You can use the shell code as is. If so, you should also read section JPEG Converter Specific Code. Those not using the shell approach or those who want to know more about the shell approach should read section low-level converter "Shell" Code to understand better what the shell code is doing.

Shell Utility Routines

In addition to calling the routines that are supplied by the converter shell client, the shell code makes available some routines that are useful to the converter. Descriptions of these functions follow.

openLowLibraryResFile

```
OSErr openLowLibraryResFile(short *fRef);
```

This routine is used to open the library resource file so that resources can be used. The caller of this routine must close the resource fork when finished. The file is opened read only.

- `fRef` is a pointer to a short that `openLowLibraryResFile` fills in with the file reference number of the library resource fork opened.

ReadWriteBackChannel

```
OSStatus ReadWriteBackChannel(PSStream *streamToClient,
 PSWriteProc writeProc,
 PSStream *streamToPrinter, PSReadProc readProc,
 unsigned char *backChannelBuffer,
 SInt32 backChannelBufferSize);
```

This routine is to be called by a converter so that any data coming up the back channel from a PostScript output device is properly passed back to the Download Manager so it can look for errors or status messages. This routine should be called regularly by a converter as it is writing data out. This routine is only for use during the `converterEmitPageData` and `converterEmitProlog` procedures (see the section Shell Usage).

- `streamToClient` is the inputStream passed to the low-level converter's `converterInitDoConvertClientData` procedure.
- `writeProc` is the write procedure on the `streamToClient` stream.
- `streamToPrinter` is the `outputStream` passed to the low-level converter's `converterInitDoConvertClientData` procedure.
- `readProc` is the read procedure on the `streamToPrinter` stream.
- `backChannelBuffer` is the data buffer passed to the low-level converter's `converterInitDoConvertClientData` procedure.
- `backChannelBufferSize` is the size of the `backChannelBuffer` as passed to the low-level converter's `converterInitDoConvertClientData` procedure.

writeLogMsg

```
OSStatus writeLogMsg(PSStream *streamOut, PSSubsection subsection,
 void *info, SInt32 stringsID,
 SInt32 msgID, Boolean isError);
```

This routine is called by a converter to log any error or warning messages which are appropriate during the data conversion.

- `streamOut` is the `outputStream` passed to the low-level converter's `converterInitDoConvertClientData` procedure.
- `subsection` is the `PSSubsection` to which the error pertains. Use `kSubAnon` if there is no appropriate subsection.
- `info` is a pointer to a structure appropriate for the subsection being reported or is `NULL`.
- `stringsID` is the ID of a `'STR#'` resource containing the message string list for the converter. The converter library resource fork is opened (and closed) by `writeLogMsg` to obtain the `'STR#'` resource, so a client need not open the library resource fork before calling `writeLogMsg`.
- `msgID` is the list number of the target message within the `'STR#'` resource referenced by `stringsID`.
- `isError` is the constant `LOGERROR` if the caller wants the message to be reported as an error as opposed to a warning. The constant `LOGWARNING` is used to report the message as a warning.

JPEG Converter Specific Code

The files in the sample code which cause this sample low-level converter to be a JPEG converter are the files sample JPEGConverterLib.r, sample JPEGConverterLib.c, and sample LangEnglish.r.

Sample 'PLGN' Resource

The file sample JPEGConverterLib.r provides the 'PLGN' resource for this converter module:

```
resource 'PLGN' (-8192,
 purgeable) {
 {
  'down', '????', "sampleJPEGConverterLib"
 }
};
```

Specifically this 'PLGN' resource indicates that the file contains one plug-in and that the type of plug-in is "down" with subtype '????'. These are the download manager plug-in types. The final piece of information indicates that the plug-in library name is sampleJPEGConverterLib. This name is the same as the name of the code fragment to load for this plug-in.

For this sample converter, the sample LangEnglish.r file contains a 'STR#' resource definition of a string list of message strings used by the sample JPEGConverterLib.c file. Your C code may require a similar 'STR#' list or other resources.

Sample Converter Description

The shell code doesn't know anything about what types of data an actual low-level converter can handle, so "shell client code" must supply a converter description. The sample code declares a new type of data MyConverterDescription which parallels the ConverterDescription data type but is concrete in the number of ConverterService structures it contains.

Here is the ConverterDescription used for the sample JPEG converter:

```
MyConverterDescription gTheConverterDescription =
{
 // signature information
 kTheConverterDescriptionSignature,  // signature always first
 kInitialConverterDescriptor,   // version data
 // ConverterType
 {
  "\pJPEG Converter", // name
  "\p",  // our real info string data comes from rsrc fork
  kMajorRev, kMinorRev, kReleaseStage, kNonRelease,
 },

 // Converter Services
 {
  kNumHandledTypes, // # of ConverterTypeInfo structures
  {
   {
    'JPEG',   // file type for JPEG data

    CANTDOWNLOAD, // priority hint - we can't
       // handle if we can't look at
       // more than the first 15
       // bytes of data to verify it
```

```
        // is JPEG data we can handle

     "\p\xFF\xD8" // the first 2 bytes of
        // JPEG/JFIF data
    },
    {
     'JFIF',   // file type for JFIF data
     CANTDOWNLOAD, // priority hint - we can't
        // handle if we can't look at
        // more than the first 15
        // bytes of data to verify it
        // is JPEG data we can handle

     "\p\xFF\xD8" // the first 2 bytes of
        // JPEG/JFIF data
    },
    {
     '????',   // file type for unknown data
     CANTDOWNLOAD, // priority hint - we can't
        // handle if we can't look at
        // more than the first 15
        // bytes of data to verify it
        // is JPEG data we can handle

     "\p\xFF\xD8" // the first 2 bytes of
        // JPEG/JFIF data
    }
   }
  }
};
```

There are a couple of points that are worth noting. First, the info field of the `converterType` field of the `ConverterDescription` is a zero-length Pascal string. The sample code takes care of filling in this field with data from sample LangEnglish.r so that the info string can be internationalized.

For our sample converter, `kNumHandledTypes` is 3, that is, there are three types of data which the converter wants to handle. The `OSTypes` that are handled are `'JPEG'`, `'JFIF'` and the unknown type `'????'`. The `ConverterTypeInfo` for each type has the same priority and `matchString` data.

The `matchString` supplied is that corresponding to the first two bytes of JPEG data. By specifying these as the first two bytes of the data stream, the Download Manager does not call this converter for files (or streams) which match the specified types but do not have these two characters at the beginning of the stream.

The priority supplied for each type is `CANTDOWNLOAD` which is the value 0. The priority value in the `ConverterTypeInfo` for each type is used only in the case where the Download Manager must assign a priority to the low-level converter but it can't call the `psLowCanConvert` routine to obtain a priority. This only happens if the input stream being downloaded cannot be repositioned, such as data that is being generated on the fly. This is never the case for file downloads from the desktop printing software in the Finder.

The sample code doesn't have to reposition the stream and this would allow it to work both with streams which allow positioning (`kPSRandomAccessStream`) and streams which do not allow positioning (`kPSSerialStream`). Unfortunately there are restrictions on the type of data that the sample code can handle; there are JPEG data types it can't handle (Progressive JPEG) and it cannot handle PostScript Level 1 output.

Because the sample JPEG converter can't always support downloading streams that begin with our `matchString` data without looking at the data further, it must advertise a priority of 0.

converterGetConverterInfoPtr Routine

The `converterGetConverterInfoPtr` routine supplied by the "shell client code" is responsible for returning a pointer to the `ConverterDescription` structure described above. As mentioned earlier, the info field of this structure is filled in with data that can be localized and lives in the low-level converter's resource fork. To access this data, the sample code opens the library resource fork, copies the appropriate string data into our structure and closes the library resource fork. The resource fork is handled in this manner because, as discussed earlier, our plug-in shared library has shared global data and this approach properly allows multiple clients to use the resource fork of the converter.

The `openLowLibraryResFile` routine is used to access the resource fork of the low-level converter. This routine is provided by the shell code to simplify access to the resource fork by the shell client code. The shell client is responsible for closing the resource fork once it is done. Note that closing the resource fork unloads any resources which have not been copied or detached from the resource fork.

converterInitDoConvertClientData Routine

This routine allows the shell client code to allocate and initialize any data that it needs to perform the conversion. One important point is that the shell code passes in a pointer to the `LanguageLevel` that reflects the job request. The shell client should update the `LanguageLevel` data pointed to by this pointer so that it reflects the minimum PostScript `LanguageLevel` that is generated by this converter. The shell code uses the data returned to generate any `%%LanguageLevel` DSC comment.

converterPeekConvert Routine

Our sample code has no need to peek at the data since it can obtain all the data it needs by reading the header of the JPEG data stream. Normally the sample code simply returns from the `converterPeekConvert` routine without doing anything.

To aid those developers who might wish to collect data during the peek phase and access that data during the conversion phase, the sample code has some conditionally compiled code to give an example of how to store private hints corresponding to data collected during the peek phase. There is also corresponding conditionally compiled code contained in the conversion phase code to access the stored private hints. Remember from the earlier Private Data Hints section that the converter might be unloaded between the `converterPeekConvert` call and the `converterInitDoConvertClientData` call, so global data cannot be used.

The reason for being careful about how private hints are stored is because any collection tags added to the hints collection may conflict with hints that are stored in that collection by the Download Manager or other portions of the code path. To overwrite such hints could produce unpredictable behavior. To avoid this problem, the collection tag value `kHintAppPrivateTag` (`'APPL'`) is reserved for third party's use. When using this tag, a developer can ensure it has a private ID value by using the signature assigned to the converter. A converter developer should register this signature with DTS, just as when developing an application.

In principle, this only allows a single piece of data to be stored by each developer. The data stored is private and need not be in any specific format. It may be convenient to have the private data itself be a collection, and the sample code demonstrates how to accomplish this.

To store a collection, one must flatten it into a handle and then store that handle. To access the stored data, one must get the collection item handle and unflatten it back into a collection. The conditionally compiled sample code does just this. It first creates a "private" collection and stores the data for a few fake "private" hints. It then flattens the private collection and stores that with the `kHintAppPrivateTag` tag value and an ID formed by using the appropriate signature. The code to access this private collection is in the sample shell client code routine `converterInitDoConvertClientData`.

converterEmitProlog Routine

The shell code client's `converterEmitProlog` routine is called by the shell code to generate the prolog code into the output stream. For the sample code, all the prolog code is contained in a small single Pascal string `psPrologL2` which is simply global data. The `PSUtilsLib` routine `psOutPStr` is called to write this Pascal string to the output stream represented by the comm `StreamInfoData` structure passed to the `converterEmitProlog` routine.

This approach is fine for converters which have small prologs, but if your converter has a large prolog, it may be preferable to store it as a resource and to load the resource and send it when needed. This has the advantage of requiring less memory during the entire conversion, since global data is in memory whenever a converter is loaded (ignoring virtual memory considerations) and resource data is only loaded upon request.

However, having the prolog in the resource fork requires each instance of a low-level converter to load its prolog rather than using the shared global data. This means that the total memory used by all instances of a low-level converter will be larger for this case. You should keep these tradeoffs in mind when deciding where to store your prolog.

After writing the prolog code, `converterEmitProlog` calls the `ReadWriteBackChannel` routine to read the output device back channel and write any data it reads back to the input stream. This allows the Download Manager to look at the data coming back from the output device and properly report PostScript errors and status information. The `ReadWriteBackChannel` routine is supplied by the shell code for use by the shell client code.

converterEmitPageData Routine

The `converterEmitPageData` routine is where the bulk of the PostScript code specific to this document is generated by the shell client code. Most of the code is pretty straightforward but there are a few comments that might be helpful.

To emit the portion of the PostScript code which parameterizes the call to the PostScript image operator, the sample code calls the `psOutFormat` routine in `PSUtilsLib`. This is one of the routines which can output data while formatting it. The call used is:

```
psOutFormat(comm, psImageDictSetup1, width,
    height, numComponents);
```

The string being output is `psImageDictSetup1`, which begins something like:

```
static const unsigned char psImageDictSetup1[] =
"\p/iwidth ^d def/iheight ^d def/components ^d def ...";
```

The use of '^d' within a string is similar to the use of '%d' in a format string for `printf`. That is, when the `psImageDictSetup1` string is scanned by the `psOutFormat` routine, it substitutes the first `^d` in the format string with the first parameter passed after the format string in its arguments. This is handled similarly for all the arguments and formatting characters in the output.

Note:
The use of "^" instead of "%" in these format strings is to avoid interference with the legitimate use of a "%" character in such format strings, since the "%" character has semantic meaning in PostScript language output.

To write the JPEG data to the output stream, the sample converter uses the `psOutBlock` routine in `PSUtilsLib`. This routine simply writes a data block of a specified number of bytes to the output stream.

For more information on the formatting routines in `PSUtilsLib`, see Appendix A.

While writing the JPEG data, the `converterEmitPageData` routine calls the `ReadWriteBackChannel` routine to

read the output device back channel and write any data it reads back to the input stream. It does this as it writes significant blocks of data to the output stream so that it detects any data coming back from the output device in a timely manner. There is no point spending time consuming computations or sending lots of data to the output device just so it can be flushed by the output device's PostScript interpreter because there was a PostScript error in the output device.

Calling `ReadWriteBackChannel` allows the Download Manager to look at the data coming back from the output device, and properly report PostScript errors and status information. The `ReadWriteBackChannel` routine is supplied by the shell code for use by the shell client code.

Use of WriteLogMsg

The sample code uses the `writeLogMsg` routine provided by the shell code. It uses this routine to generate error or warning messages that are available for processing by the Download Manager or the application client which invoked the Download Manager. In addition, these messages can be saved into a log file. See the Tips section for additional information about logging that might be useful for debugging a low-level converter.

Back to top

low-level converter "Shell" Code

This section describes the operation of the shell code in more detail. In principle, only those who want a deeper understanding of the shell code or who need to modify it need to read this section. In practice, since the shell code is compiled into your low-level converter, you might want to understand it better even if you don't plan to modify it.

The biggest limitation of the shell code as implemented for the sample JPEG converter is that it is currently hard coded to generate one page of output. The shell code's `psLowGetStreamInfo` routine explicitly uses 1 for the number of pages it reports, regardless of whether EPS output is being generated. It treats the bounding box data as if there is only one page being generated. It also uses 1 for the number of pages it generates for the `%%Pages: comment` in the document header and for the page number it generates for the `%%Page: comment` on the first (and only) page. It only calls `converterEmitPageData` once and doesn't pass the current page number since the assumption is that there is only one page. In addition, there are several places where a pointer to the current page number is passed to the `doOutputPosition` routine (discussed in the section DSC Comments and Feature Code).

psLowGetStreamInfo

This routine is implemented in a generic fashion since the only data required from the shell client code is the type of data that is actually contained in the file. Since the data in the stream is opaque to the caller of the Download Manager, the low-level converter which can perform the download is the only entity that can authoritatively determine the data type. The shell code calls the shell client code's `converterGetConverterDocType` routine to determine what the data type is. The shell client code can scan the data to determine this, or, in the case of the sample JPEG converter, it simply returns the data type since there is only one type of data that the sample converter can handle. Since the only way a low-level converter's `psLowGetStreamInfo` - and therefore its `converterGetConverterDocType` routine - can be called is if the converter has already agreed that it can handle the data through the `psLowCanConvert` routine, it doesn't have to scan the data unless it supports multiple data types.

The shell code determines whether an EPS or PostScript language download job is requested. If it is EPS, the shell code knows the number of copies is always 1; otherwise, it looks for the job hint which indicates the number of copies to generate and returns that value. If the job is not EPS, it calls the routine `psIsJobManualFeed` in the `FeatureUtilsLib` library to determine if the hints collection corresponds to a manual feed job. If the job is EPS, it is never a manual feed job.

As described above, the shell code always reports 1 for the number of pages.

psLowDoConvert

The shell version of this routine takes care of the bulk of the generation of the DSC comments, the feature code, and the invocation of the shell client code's routines for generating the PostScript output into the stream.

It begins by allocating a buffer for reading the back channel data that might be returned by the output device back to the host. It does this so that both the shell client and the shell code itself can use this buffer to read the back channel.

The shell code calls the routine psSetupStreamInfoData to obtain a StreamInfoData structure that can be used with the output routines contained in the PSUtilsLib library. The resulting comm variable is passed to those shell client routines that are likely to emit code. If there are other routines that need the comm structure, the shell code must be modified accordingly.

Back to top

DSC Comments and Feature Code

The shell code adds a number of hints that, on the surface, don't seem to be used anywhere. These hints include: kHintEPSBBoxTag, kHintAppNameTag, kHintClientNameTag, and kHintClientVersionTag. These hints are used by the code which generates the feature invocations; that code is contained in the FeatureUtilsLib library. For example, when a cover page is generated, some of these hints are used to obtain data to fill in cover page information.

The shell code emits a number of DSC comments into the stream by using the doOutputPosition routine. This routine combines the generation of the DSC comments into the output stream (with the appropriate tagging) and the request for various features.

The routine doOutputPosition is actually a macro which results in a call to the routine OutputPosition. The OutputPosition routine uses the routines psWriteSubsectionFeature, psOutFormatPosition, and psOutFormatPositionInfo in a stylized fashion that is appropriate to discuss here. Here is the routine:

```
static OSStatus OutputPosition(StreamInfoData comm,
 Collection hints, const SubsectionStr *subsectionStr,
 void *info, Boolean isNotEPS)
{
 OSStatus err = noErr;

 err = psWriteSubsectionFeature(comm, hints,
  subsectionStr->subsection, info,
  kBeforeSubsection, isNotEPS);
 if(!err){
  if(info)
   err = psOutFormatPositionInfo(comm,
     subsectionStr, info);
  else
   err = psOutFormatPosition(comm, subsectionStr);
 }
 if(!err)err = psWriteSubsectionFeature(comm, hints,
   subsectionStr->subsection, info,
   kAfterSubsection, isNotEPS);
 return err;
}
```

The OutputPosition routine first calls psWriteSubsectionFeature with the correct subsection and with the value kBeforeSubsection. It then writes the comment by making a call to psOutFormatPosition or psOutFormatPositionInfo depending on whether the info data passed in is NULL or not. Finally, it calls psWriteSubsectionFeature with the correct subsection and with the value kAfterSubsection.

psWriteSubsectionFeature is a routine in FeatureUtilsLib that uses its knowledge of job feature requests contained in the hints collection passed to it, and combines that knowledge with the information about what portion of the

document is currently being generated. When this routine is called, it is passed a PSSubsection which indicates which DSC comment is going to be written or was just written. A psSubsectionLocation is also passed to psWriteSubsectionFeature and indicates whether this call to psWriteSubsectionFeature is before or after the DSC comment which is being written.

By passing this information about the document structure to psWriteSubsectionFeature, it is able to intelligently generate the feature code needed at the appropriate point of the job stream. What feature code to generate depends on the features requested in the hints collection passed to it. For example, when a call is made to psWriteSubsectionFeature with the subsection value kSubPSAdobe and psSubsectionLocation value kBeforeSubsection, psWriteSubsectionFeature knows to determine whether a cover page should be emitted and if so, emits the cover page code into the stream. The call returns and the caller normally generates the %!PS-Adobe-3.0 DSC comment. It then calls psWriteSubsectionFeature with the subsection value kSubPSAdobe and psSubsectionLocation value kAfterSubsection and psWriteSubsectionFeature writes nothing to the output stream.

This stylized way of using psWriteSubsectionFeature allows feature code to be generated corresponding to the feature requests in the hints collection. The feature invocation code is generated into the job stream in the appropriate place, as long as the caller gives the psWriteSubsectionFeature routine a chance to write its feature data at the appropriate points in the DSC job stream. The shell code does this so that neither the shell code nor the shell client code need to know anything about the features that the job requires. At the same time, the shell and its clients have to do little work to support those features. There is nothing that says a low-level converter must use the feature generation ability of the code in FeatureUtilsLib. However, by using this library as shown in the sample code, the user will obtain the requested print time features.

FeatureUtilsLib is described in more detail in Appendix C.

The psOutFormatPosition and psOutFormatPositionInfo functions mentioned above are available in PSUtilsLib and are used to tag the output written into the stream so that the Download Manager or its callers who wish to track the progress of the job can do so by looking at the tags. This 'tagged' data is actually part of the stream itself in the form of the PSPosition structure, which is part of a PSSerialStream structure.

The psOutFormatPosition and psOutFormatPositionInfo routines are passed the string data to be written and the PSSubsection tag together in a SubsectionStr structure. This allows the data written to be easily marked with the appropriate tag. The only difference between the two calls is that a call to psOutFormatPositionInfo supplies a (void *) pointer to some information data that is put into the info field of the PSPosition structure on the stream. The call to psOutFormatPosition is equivalent to a call to psOutFormatPositionInfo with the info parameter set to NULL. More information on these functions is available in Appendix A.

Auto Scaling

It is important to point out again that the shell code does auto scaling so that the page of output is centered and, if it would not fit on the page, is scaled to fit on the page. The orientation of the output depends on the dimensions of the printed sheet and the bounding box of the data drawn. The drawing is oriented on the page so that the longest dimension of the bounding box is aligned with the longest dimension of the paper.

This type of scaling may not be appropriate for all types of converters. It is implemented as a call to psWriteSubsectionFeature with the kSubAutoScaling PSSubsection value. Clearly this data does not correspond to a DSC comment, and there is no DSC comment written before or after this call to psWriteSubsectionFeature. This is a bit different than the use of psWriteSubsectionFeature described above and is not associated with any DSC data written.

This is done in this manner for two reasons. First, not all users of psWriteSubsectionFeature are interested in emitting the auto scaling code. In addition, the appropriate place to emit the auto scaling code depends on the PostScript code being generated for drawing a given graphic. By not tying the auto scaling code to a specific DSC section, the caller decides whether to include the code and if so, it can decide exactly where to include it.

Device Adjust Matrix

The shell code generates a device adjust matrix adjustment which reflects the *DeviceAdjustMatrix value (if any) in the PPD file representing the target output device. This is done as a call to psWriteSubsectionFeature with the

kSubDeviceAdjustMatrix PSSubsection value. The shell code does not write a DSC comment before or after this call to psWriteSubsectionFeature. Again, this is a bit different than the use of psWriteSubsectionFeature described above, and is not associated with any DSC data written. When psWriteSubsectionFeature writes any device adjust matrix code, it generates %%BeginFeature and %%EndFeature around the code, just as it does for other PPD feature code that it generates.

Similar to the handling of kSubAutoScaling, the kSubDeviceAdjustMatrix PSSubsection has no connection to a specific point in the structured document job stream. The appropriate place to emit the device adjust matrix code depends on the PostScript code being generated for drawing a given graphic. By not tying the device adjust matrix code to a specific DSC section, the caller decides whether to include the code and if so, where exactly to include it.

Note that the shell emits PostScript code surrounding the invocation of psWriteSubsectionFeature with the PSSubsection value kSubDeviceAdjustMatrix. This is done so that if there is no device adjust matrix code generated, there is no adjustment. The code to use a device adjust matrix properly is document dependent and therefore the program which uses the matrix needs to decide how to use it. The feature code merely inserts the matrix (if there is one) in the stream.

Back to top

Tips

Converter Priorities

The Download Manager favors external converters over internal converters of the same priority. This means that if the sample JPEG converter is placed in the "Printing Plug-ins" folder it is favored over that built into PrintingLib. This is fine for looking at the sample converter; however, to use the JPEG converter built into PrintingLib, the sample JPEG converter must be removed from the "Printing Plug-ins" folder.

'PLGN' Resource Editing

While not strictly necessary, a 'PLGN' resource ResEdit template is in PrintingLib 8.6 and later. To look at the 'PLGN' resource in PrintingLib, just open it up. If you want to look at the 'PLGN' resource in the low-level converter you build, copy the appropriate 'TMPL' resource into that converter and then open the 'PLGN' resource.

Caching Issues

The Download Manager resolves aliases placed in the "Printing Plug-ins" folder. It is convenient to put an alias to a low-level converter in the "Printing Plug-ins" folder during converter development. If there isn't already a "Printing Plug-ins" folder in the Extensions folder, the Download Manager will create one automatically when it is called for the first time. One way to cause this to happen by dragging any document onto a desktop printer created by LaserWriter 8 when using Mac OS 8.5 or later.

To improve performance, the Download Manager caches both the list of converters in the "Printing Plug-ins" folder and the ConverterDescription information it obtains from each low-level converter. The Download Manager uses the folder modification date of the "Printing Plug-ins" folder to determine whether it needs to update its cached list of converters. This can be an issue during the development of a low-level converter.

If the 'PLGN' resource or the ConverterDescription information in a plug-in file changes, you want the Download Manager to notice and take the new information into account. However, the system software updates the folder modification date only when items are added or removed from a folder. If an item is edited in place, the system does not change the folder modification date. This means that editing or rebuilding a plug-in file directly in the "Printing Plug-ins" folder, the folder modification date may not change. This also applies to an alias that points to a plug-in file that is created in another directory.

This "problem" only manifests itself when the 'PLGN' resource or ConverterDescription is changing, specifically, very early on in the development of a converter. Until the plug-in recognized by the Download Manager and the ConverterDescription has been finalized, the new converter should be manually copied it to the "Printing Plug-ins" folder.

Note:
This is not a problem for users since there is no appropriate way to edit a plug-in file.

Initial Software Development

Getting Your Converter Seen by the Download Manager

Until the `'PLGN'` resource is correct and all required symbols are correctly exported, the Download Manager will never call a low-level converter. Once these requirements have been met, the Download Manager will call the `psLowGetConverterInformation` of the low-level converter when the user drags and drops a file onto a desktop printer created by LaserWriter 8 (in Mac OS 8.5 and later) and the modification date of the "Printing Plug-ins" folder has changed since the last drag and drop. It is wise to start converter development by making sure that the Download Manager detects the new low-level converter.

The easiest way to do this is to put a breakpoint on the converter's `psLowGetConverterInformation` routine and dropping a document onto a desktop printer created by LaserWriter 8. If this fails, either the `'PLGN'` resource is not correctly formed or the low-level converter does not export all the required functions. In order to retest, make sure the "Printing Plug-ins" folder modification date has changed before.

Getting Your Converter Called For Your Data Types

Once the converter is seen by the Download Manager, the next thing is to make sure that the converter is being given a chance to convert all the files which have match the data types and `matchString` entries in the `ConverterDescription`. This is the process of getting the `ConverterDescription` correct for a low-level converter. The simplest way to make sure a converter is getting asked about all the data types (and `matchStrings`) it expects is to put a breakpoint on the `psLowCanConvert` routine and verify that this routine is being called by the Download Manager. Once the `psLowCanConvert` routine is called as expected, you are ready to do the real work of implementing all the routines and converting the data.

If the converter's `psLowCanConvert` routine isn't getting called as expected, but `psLowGetConverterInformation` is being called, the culprit is the `ConverterDescription` being returned by `psLowGetConverterInformation`.

Note:
Once your converter is being properly called for all your data types, the caching issues can usually be ignored for the rest of your software development.

Logging

A low-level converter (and the Download Manager itself) can tag data that it writes to a stream by setting `PSSubsection` values in the `PSPosition` structure that is part of the stream (see Appendix A for more information about the streams the Download Manager uses). When a low-level converter uses the `PSSubsection` values `kSubLogErrorData` and `kSubLogWarningData`, it is passing error or warning messages back to the Download Manager. The Download Manager gives its clients an opportunity to report these messages to the user.

The Download Manager has the ability to write these error and warning messages to a log file. This ability is turned off in the version of `PrintingLib` shipped with the system software, but it easily can be enabled and tailored slightly by using ResEdit or Resorcerer to edit the `PrintingLib` file.

Using Logging

Developers are encouraged to use the `PSSubsection` value `kSubLogWarningData` to generate warning messages that are useful to sophisticated users. For example, if, in the middle of a conversion, a converter discovers that the data may have a problem but the problem isn't fatal, that information could be reported with a warning message. Of course, fatal

errors should be reported using the PSSubsection value kSubLogErrorData.

In addition, it may be useful to add warning messages as part of debug builds of a low-level converter. This allows you and your testers to look at a trace of what is happening during the execution of your converter. This may be useful as a supplement to the standard debugging strategies of setting breakpoints or using debug strings.

Enabling Logging

> **Note:**
> As always work with a copy and, to enable logging, you should reboot your computer after editing the PrintingLib file.

To turn on logging, edit the 'PRF2' resource in PrintingLib (Version 8.6 and later). There is a bit labeled "Generate Log File for Download Manager Errors and Warnings" which is off by default. Turn this bit on and save your changes. Reboot. From this point on, logging is enabled.

The logging ability does have a bit of flexibility that might be useful to some developers or sophisticated users. It can be configured slightly by editing the 'LOGD' resource in PrintingLib. Open the 'LOGD' resource and you'll see a number of editable items:

- The maximum log file size (default: 32000 bytes).
- How much of the existing log to preserve when the log file size exceeds its maximum (default 4000 bytes).
- The Creator and OSType of the log file by the Download Manager (default: MPW text type).
- The name of the Log file (default: "Download Manager Log").

When logging is enabled, the log file with the name specified by the 'LOGD' resource is created in the "Printing Prefs" folder in the Preferences folder.

Back to top

Summary

This Technote describes how to write a low-level converter for use with the Download Manager, part of LaserWriter 8 and PrintingLib, Version 8.6 or later. If your application supports or defines a file format which could easily be converted to PostScript without launching the application, you should consider writing a low-level converter to support printing files of that format directly when the user drags a file to a desktop printer. This allows for faster printing since no application needs to be launched in order to print. Since low-level converters output PostScript directly, writing a converter can offer you the opportunity to optimize printing of your file formats on PostScript output devices.

References

Technote 1169: The Download Manager

Technote 1170: The Printing Plug-ins Manager

Inside Macintosh: QuickDraw GX Environment and Utilities

Inside Macintosh: PPC System Software (CFM)

Change History

Originally written in April 1998 by David Gelphman and Ingrid Kelly

Revised in June 1999 by Dave Polaschek

Back to top

Appendix A: Useful PSUtilsLib Routines and Structures

Low-level converters write their generated PostScript data to procedures passed in a structure of type PSStream. These procedures can be called to read from a data source or to write to an output device or another data consumer. Because writing to streams is very common in the operation of both conventional LaserWriter 8 driver printing and the operation of the Download Manager low-level converters, the PSUtilsLib library (contained in PrintingLib) exports a number of useful routines which handle many of the details of writing to streams. This Appendix focuses on the details of writing to streams as well as documenting some of the routines available in PSUtilsLib and their usage.

PSStreams are discussed in further detail in Technote 1169, "The Download Manager." In addition, the PSStreams.h header file contains the definition of the PSStream data type as well as the routines described in this Appendix.

PSStream Structure

The PSStream structure describes a number of stream types. The important stream types for a low-level converter are those of type PSSerialStream and PSRandomAccessStream. The PSRandomAccessStream stream type allows read access to data in a random way; the stream allows the caller to position the stream mark randomly. This stream is used to represent files or data that can be accessed as if it were in a file. This type of stream is typically used as an input stream to a low-level converter. Other than the random access nature of these streams, they are identical to the PSSerialStream so the remainder of this discussion will be about the PSSerialStream type of PSStream.

The PSSerialStream is defined as follows:

```
typedef struct PSSerialStream{
 PSWriteProc  write;
 PSReadProc  read;
 UInt32  reserved;
 PSPosition  pos;
}PSSerialStream;
```

The write proc of a PSSerialStream is used to write PostScript data to a consumer of the data. The write proc of an output stream typically writes data to a PostScript output device or data file. The write proc (if it exists) on an input stream writes data back to the Download Manager or similar client for further processing. For example, it is appropriate to write data read from an output stream back to the input stream so that the Download Manager can handle status or other data returning from the back channel of an output device. It is important to test that the write proc is not NULL before calling it.

The read proc of a PSSerialStream is used to read data from that stream. The read proc of an input stream reads the data from the input stream. For a low-level converter, this is the data to convert. The read proc (if it exists) of an output stream represents data coming back from a PostScript output device. It is important to test that the read proc is not NULL before calling it.

The reserved field in the PSSerialStream structure is currently unused by a low-level converter.

The PSPosition structure in the PSSerialStream communicates structural information about the data being written to a stream. This is discussed in detail in the next section.

PSPosition Structure

The PSPosition structure allows generators of PostScript output to communicate structural information about the data they are writing. When generators of PostScript output properly use the PSPosition structure, it allows software

clients to have knowledge of the data being written, without them having to parse the PostScript data itself. An example of this is the way the LaserWriter 8 driver reports status during printing by looking at the PSPosition data written to the output stream by the PrintingLib routines which convert QuickDraw drawing into PostScript data. Another example is the status that the Download Manager and its clients report as a low-level converter generates its PostScript data.

The PSPosition structure is defined as:

```
typedef struct PSPosition{
PSSection section;
PSSubsection subsection;
void *info;
SInt32 id;
}PSPosition;
```

The section field is of type PSSection and contains the identification of what "major" part of the job is in progress. The values of this field can be kSectAnon, kSectQueryJob, kSectCoverPage, kSectJob, and kSectPeek. These correspond to the different parts of the job, as controlled by the Download Manager and this field is filled in by the Download Manager, not by the low-level converter.

The subsection field is of type PSSubsection and is used to describe the details of the PostScript output corresponding to the data write call. PSSubsection values typically correspond to Document Structuring Conventions (DSC) data but there are additional values which suit some specialized needs.

The info field is either a NULL pointer or a pointer to data whose type is defined for the PSSubsection value in the subsection. The data (if any) pointed to by the info value coincides with the data being written to the output stream. For example, when calling the write routine with the data "%%Pages: 4", the caller would put the PSSubsection value kSubPages into the subsection field of the PSSerialStream and the info field would point to an SInt32 with the value 4. See the header file PSStreamInfo.h for the list of PSSubsection values and the proper data type for the info of each PSSubsection.

The ID field is an SInt32. This is used by generators of the PostScript output to generate output for a given subsection over a series of writes, yet still identify the data as one conceptual block of data. This is done by performing the consecutive writes with the same subsection, info, and ID values. When the data being written corresponds to a new subsection, then the ID value is incremented. Doing writes in this fashion allows software clients looking at the structural data to notice when the PSPosition data may have changed without having to look at any other fields in the structure. For example, a client (such as the Download Manager) monitoring the position information being written to the stream has a test like:

```
if(jobstatus->lastPosId != stream->u.ps.pos.id ){
 ... process the new position we are now seeing
 ...
 // update our the last position we saw
 jobstatus->lastPosId = stream->u.ps.pos.id;
}
```

Simple Example of Writing to a Stream

Here is a simple code example to bring together the basic ideas presented on streams. The data is hard coded into this routine to improve readability.

```
#include "DownloadMgrLib.h"
#include "PSStreams.h"

OSStatus writePages(PSStream *streamOut)
{
 OSStatus err = noErr;
 PSSerialStream *stream;

 if(streamOut->type == kPSRandomAccessStream)
  stream = &(streamOut->u.file.serialStream);
 else{
  if(streamOut->type == kPSSerialStream)
   stream = &streamOut->u.ps;
  else{
   // we don't know that type of stream!
   err = errCantHandleThisDownloadData;
  }
 }

 if(!err && stream->write){
  SInt32 pages = 4;
  unsigned char *formatString = "\p%%Pages: ";

  // the subsection reflects the fact that
  // we are writing kSubPages
  stream->pos.subsection = kSubPages;

  // the info field is a pointer to the number of pages
  stream->pos.info = &pages;

  // distinguish this write from any previous
  stream->pos.id++;
  // now go ahead and write the '%%Pages: ' portion
  // of the comment
  err = stream->write(streamOut, formatString +1,
       formatString[0]);

  // now go ahead and write the value of the number
  // of pages with the SAME id since it is part of
  // the same DSC data we are emitting
  if(!err){
   Str15 pagesStr;
   NumToString(pages, pagesStr);
   err = stream->write(streamOut,
     (unsigned char *)pagesStr + 1,
     pagesStr[0]);
  }

  // now write the newline with the SAME id
  if(!err){
   err = stream->write(streamOut, "\r", 1);
  }
```

```
  // reset the PSPosition data after our write call
  stream->pos.subsection = kSubAnon;
  stream->pos.info = NULL;

  // we must bump the id so that consumers of this
  // stream will know that we are done with the
  // write of the Pages comment when the next write
  // is done.
  stream->pos.id++;         }

 return err;
}
```

Note:
The id field of the PSPosition structure on the stream is updated before the first write of the Pages comment and after the write of the last portion of the Pages comment. The last id increment is done so that we ensure that any following write to the stream is distinguished from this write of the Pages comment. This is more than a safety measure since many of the stream output routines do not modify the PSPosition structure of the stream. Therefore, after our write, the stream should already reflect a new id to distinguish future writes from the one just done.

Useful Stream Output Routines

Generating PostScript output for a given print job typically involves emitting both constant data such as the %%Pages comment, as well as variable data such as the SInt32 value for the number of pages as in the example above. Sometimes the data needs to be formatted differently depending on the characteristics of the output communications channel. The most obvious example of this occurs when generating PostScript string data since there needs to be quoting of various characters, depending on whether the channel supports the full range of binary data. The need to supply the PSPosition information while generating output adds an additional requirement when generating output.

The PSUtilsLib library built into PrintingLib has routines which make generation of PostScript output significantly simpler. PSUtilsLib contains routines that make it simple to generate formatted output with and without positional information.

Relevant Structures

Before introducing the output routines, there are a couple of relevant data types that must be introduced first.

StreamInfoData

The StreamInfoData type is a pointer to an opaque data structure that is passed to the PSUtilsLib stream output routines. This opaque structure contains information about the communications channel which enables the stream formatting routines to generate proper PostScript output. There are routines for creating and disposing of this structure.

```
typedef struct StreamInfo *StreamInfoData;

OSStatus psSetupStreamInfoData(StreamInfoData *comm,
   PSStream *PSStreamP, Collection hints);
```

psSetupStreamInfoData allocates and initializes a StreamInfoData structure corresponding to the PSStreamP

and the hints collection. It consults the hints collection for hints indicating the capabilities of the communications channel (see Appendix B). The resulting StreamInfoData can then be passed to the stream output routines described below to write to the stream represented by PSStreamP and generate output properly formatted for that communications channel.

```
OSStatus psDisposeStreamInfoData(StreamInfoData *comm);
```

psDisposeStreamInfoData disposes of the StreamInfoData structure that was created and returned from psSetupStreamInfoData. Upon return of this routine, *comm is NULL.

SubsectionStr

When generating PostScript output that is to be tagged with a given PSSubsection value, it is useful to group the PostScript output string together with an associated PSSubsection value. The data structure SubsectionStr gathers these pieces in one place. The definition of SubsectionStr is:

```
typedef struct SubsectionStr{
 StringPtr format;
 PSSubsection subsection;
}SubsectionStr;
```

An example of a SubsectionStr for generating the %%Pages DSC comment would be:

```
const SubsectionStr psPages = {"\p%%Pages: ^d\r",kSubPages};
```

The format field of the SubsectionStr is a Pascal string that may contain formatting data. In the above example the format uses the ^d formatting marker. This will be described shortly.

Formatting Output Routines

The psOutFormat routines and its structured equivalents described below allow straightforward use of output formatting similar to the printf routine in the standard C library. Because the '%' character is a significant character in PostScript data, these routines use the '^' character as the format marker character.

```
OSStatus psOutFormat(StreamInfoData comm,
     ConstStr255Param format, ...);
```

The supported formats are:

- ^b pass in a long and output "true" or "false"
- ^d pass in a long and output in decimal format.
- ^f pass in a 16.16 fixed number and output in decimal with up to 3 places past the decimal.
- ^F pass in a 16.16 fixed number and output in decimal with up to 4 places past the decimal.
- ^H pass in a long and the long div 2 is output with a possible .5 (or you can think of it as a 31.1 Fixed-point number)
- ^s pass in a pointer to a Pascal String. For use when generating PostScript strings, i.e., (^s)
- ^S same as ^s, but with control and extended ASCII characters always quoted. Typically used for DSC comments which are always in the range `0x20 - 0x7F`
- ^z same as ^s, but specified with explicit length (call with string pointer and long length).
- ^Z same as ^s, but generate (..) or <..> depending on which one takes the least space.
- ^i pass in a short and it is output in decimal format.
- ^p same as ^s, it outputs a Pascal string.

```
OSStatus psOutFormatPosition(StreamInfoData comm,
    const SubsectionStr *format, ...);
```

`psOutFormatPosition` is just like `psOutFormat` except that it takes a pointer to a `SubsectionStr` structure rather than a format string. The `SubsectionStr` structure provides both a format string and a `PSSubsection` value for that format string that will be passed to the stream's output routine to identify the type of PostScript that is being written. `psOutFormatPosition` first inserts the `PSSubsection` value into the subsection field of the `PSPosition` in the stream and stores a `NULL` into the info field in the stream's `PSPosition` structure. It then writes the formatted output to the stream. This routine takes care of ensuring that the `PSPosition` data is handled appropriately, i.e., in a similar manner to that shown above in the section Simple Example of Writing to a Stream.

```
OSStatus psOutFormatPositionInfo(StreamInfoData comm,
   const SubsectionStr *format, void *info, ...);
```

`psOutFormatPositionInfo` is just like `psOutFormatPosition` except the info value passed to this routine is stored in the `PSPosition` structure in the stream that is passed to the write routine prior to the write. The info pointer provides additional information to the PostScript positional information provided by format. After `psOutFormatPositionInfo` returns, the info field of the steam's `PSPosition` structure is null.

Simple Formatted Example

```
OSStatus writeFormattedPages(StreamInfoData comm)
{
 OSStatus err = noErr;
 const SubsectionStr psPages = {"\p%%Pages: ^d\r", kSubPages};  SInt32 pages = 4;

 err = psOutFormatPositionInfo(comm,
   // the format
   &psPages,

   // now the info. For the kSubPages it is a
   // pointer to an SInt32
   &pages,

   // now the data to satisfy the format. The ^d
   // takes this long and writes the output
   pages);

 return err;
}
```

Additional Formatting Routines

```
OSStatus psOutHexBlock(StreamInfoData comm, Byte *block,
    long nBytes, short *linePos);
```

`psOutHexBlock` writes `nBytes` from block to the stream represented by `comm` using the hex encoding technique, regardless of the channel characteristics. The hex data generated is wrapped to avoid excessively long lines. `*linePos` represents the current length of the line and is initially passed as 0. Upon return, `*linePos` represents the length of the current line. Each sequential call to `psOutHexBlock` should pass in the value returned from the previous call. `psOutHexBlock` is useful when generating image data when the output channel does not support binary data and `ASCII85` is not appropriate.

```
OSStatus psOutBlock(StreamInfoData comm, const void *block,
       long nBytes);
```

`psOutBlock` writes `nBytes` of data from block to the stream represented by comm without any additional processing. `psOutBlock` is useful for emitting binary image data or other output that requires no additional formatting.

```
OSStatus psOutString(StreamInfoData comm, Byte *str, long length,
     Boolean quoted, short *linePos);
```

psOutString writes length bytes of data pointed to by str assuming that it is going to be inside a PostScript string. This function performs the quoting necessary for the channel and does line breaks as necessary. If quoted is true, then bytes outside the printable ASCII character set are always quoted, regardless of the communications channel characteristics. If quoted is false, then bytes outside the printable ASCII character set are quoted according to the needs of the communications channel. The string data generated is wrapped to avoid excessively long lines. *linePos represents the current length of the line and is initially passed as 0. Upon return, *linePos represents the length of the current line. Each sequential call to psOutString should pass in the value returned from the previous call.

```
OSStatus psOutPStr(StreamInfoData comm, ConstStr255Param pstring);
```

psOutPStr writes the Pascal string pstring to the stream represented by comm. There is no quoting or formatting done.

Back to top

Appendix B: Available Job Queries

Low-level converters can specify printer queries to help them to generate optimal PostScript data for the target output device. The low-level converter uses its psLowAddConverterQueries routine to add hints to a query collection that can be used by the Download Manager to query information about the target output device. This Appendix describes each available query hint in detail. The header file Hints.h contains the actual tag and ID values as well as the definition of any structures that are used to store query results.

Communications Channel Queries

The query hints kHintTransparentChannelTag, kHintTransparentChannelId, kHintEighthBitTag, and kHintEighthBitId specify queries related to the capabilities of the communications channel. Whether the communications channel can support full binary data or only a subset of such data is important to generators of PostScript code. Generating full binary output is much more efficient but it is not acceptable if the communications channel does not support it!

Normally a low-level converter will add both of these hints to the query collection with default values of false to specify that the Download Manager supply the appropriate query for the channel characteristics. The value for these hints after the query determines the channel characteristics.

If the value of the hint with tag value kHintEighthBitTag and ID value kHintEighthBitId is true, the output stream supports the data range 0x80-0xFF inclusive. If the value is false, the PostScript output stream generated by the low-level converter should not contain these byte values.

If the value of the hint with tag value kHintTransparentChannelTag and ID value kHintTransparentChannelId is true, the output stream supports the data range 0x00-0x1F inclusive. If the value is false, the PostScript output stream generated by the low-level converter should not contain these byte values.

Note:
A `StreamInfoData` structure, described in Appendix A, is configured by these hints. Consequently, the relevant stream output formatting procedures described in Appendix A then know how to format PostScript output properly for the stream.

Output Device Characteristics

There are several queries available to allow a low-level converter to determine the inherent capabilities of a given output device. Knowledge of this information typically enables the generation of much more efficient PostScript output.

PostScript Language Level

To query for the PostScript language level of the output device, a converter adds the hint with tag value `kHintLanguageLevelTag` and ID `kHintLanguageLevelId`. The value returned is an `SInt32`. The following enum describes the currently defined values.

```
enum PostScriptLevels{
 /// L2 compatible
 Level2and3 = -3,
 /// L1 compatible
 Level1and2 = -2,
 /// unknown level
 UnknownLevel = -1,
 /// other level
 OtherLevel = 0,
 /// level 1
 Level1 = 1,
 /// level 2
 Level2 = 2,
 /// level 3
 Level3 = 3
};
```

Positive values indicate a specific PostScript language level, for example the value 2 means that the target output device supports language level 2. In this case there is no need to generate output compatible with a PostScript level 1 output device and use of level 3 (or later!) operators will generate errors.

Negative values returned from this query are associated with either an Unknown response or indicate a request for generating output compatible with a given minimum language level. If the language level returned is `UnknownLevel` or `Level1and2`, then typically a low-level converter should generate output compatible with PostScript language level 1. Such output may use language level 2 or language level 3 features but it must do so in a way that also executes properly on a language level 1 output device. If the value is `Level2and3`, this indicates that the generated PostScript must be compatible with a language level 2 output device. Such output may use PostScript 3 features but must do so in a way that also executes properly on a language level 2 output device.

Color Output Device

Prior to generating sampled image data, it may be useful to know whether the target output device supports color. If it does not support color then in many cases it may be more efficient to downsample any RGB or CMYK data into grayscale data as part of generating the PostScript language output. There are two queries which relate to the output device's ability to produce color output.

The query specified with tag value kHintColorDeviceTag and ID value kHintColorDeviceId queries for whether the output device is known to support color output. The value returned from this query is of type TriState.

```
 enum TriState{
 kTriFalse = 0,
 kTriTrue,
 kTriUnknown
};
typedef enum TriState TriState;
```

If the value returned is kTriTrue then the output device supports color. If the value returned is kTriUnknown then it is unknown whether the output device supports color. A low-level converter should not do any downsampling of color data to grayscale for either of these cases.

If the value returned is kTriFalse then the output device does not support color and the color separation query (just below) should be consulted to determine whether the output device is configured to generate color separations. If a black and white output device is generating color separations then color data should be emitted so that the separations are generated properly.

The query specified with the tag value kHintColorSepTag and ID value kHintColorSepId queries for whether the output device is known to be configured to generate color separations. The value returned from this query is of type TriState.

If the value returned is kTriTrue then the output device is generating color separations. If the value returned is kTriUnknown then it is unknown whether the output device is generating color separations. A low-level converter should not do any downsampling of color data to grayscale for either of these cases.

If the value returned is kTriFalse, then the output device is not generating color separations. In this case it would only be appropriate to generate downsampled grayscale data if the output device is not generating color separations and is known to not support color output.

Device Resolution

The query specified by the tag value kHintPrinterResTag and ID value kHintPrinterResId queries for the current device resolution at the time of the query. The data returned from the query is of type PSResolution.

```
struct PSResolution{
 long x;
 long y;
};
typedef struct PSResolution PSResolution;
```

The returned resolution data is in dots per inch (dpi) and may differ in X and Y. If the resolution is unknown, a value of -1 is returned for both X and Y. Note that generally it is a mistake to use device resolution data when generating PostScript output since doing so hampers a given output device's ability to produce the best quality output.

Printer Resources

TrueType Rasterizer

The query specified by the tag value kHintTTRasterizerTag and ID value kHintTTRasterizerId queries for the support level available for TrueType fonts. The value returned for this query is a long with the following values defined:

```
enum TTRasterizerType {
 kTTRasterizerUnknown = 0,
 kTTRasterizerNone = 1,
 kTTRasterizerAccept68K = 2,
 kTTRasterizerType42 = 3
};
```

If the value returned is kTTRasterizerType42 this indicates that the target output device has built-in support for FontType 42, i.e., TrueType, fonts. If the value returned is kTTRasterizerAccept68K, this indicates that the output device has no built-in rasterizer but it can accept a downloaded rasterizer. If the value returned is kTTRasterizerNone this indicates that the output device has no support for TrueType fonts and a rasterizer cannot be downloaded. A value of kTTRasterizerUnknown means that the availability of a TrueType rasterizer in the target PostScript output device is unknown.

Fonts

Low-level converters can request a query for a specific list of fonts or request the entire list of fonts available in the target output device. Both of these font queries are specified with the hint tag kHintIncludeFontsTag with the ID value kHintIncludeFontsId. The initial data contained in this hint determines the type of query. The data is a PSFontHandling structure, defined as:

```
typedef struct {
 long tag;
 unsigned char name[1]; //packed array of names as PStrings,
      //length 0 indicates end of list
}PSFontHandling;

and the following tag values are defined:
enum{
 kIncludeNoFontsOtherThan,
 kIncludeAllFontsBut
};
```

If the tag field of the PSFontHandling structure is kIncludeAllFontsBut, the query is for the complete list of fonts (the equivalent of the *?FontList query from the PPD file). For this flavor of the font query, a converter should pass in one font name whose length is zero. Upon return of the query, the name field will be a packed array of Pascal strings corresponding to the fonts built into the output device. This list of names will be terminated with a zero-length Pascal string.

If the tag field of the PSFontHandling structure is kIncludeNoFontsOtherThan, the query is for a specified list of fonts (the equivalent of the *?FontQuery query from the PPD file). For this flavor of the font query, the list of fonts to query for should be in the name field of the structure. The list is a packed array of Pascal strings and is terminated with

a zero-length Pascal string. After the query, the name field is a packed array of Pascal strings corresponding to the fonts from the query list which were not available, i.e., the fonts available in the output device are removed from the list. Again, this list of names is terminated with a zero length Pascal string.

Note:
Requesting either type of query can produce results in the other form. For example, a request for all fonts can result in a list of fonts which are not available. A request for the availability of a list of fonts can result in a list of all fonts. This means, for example, that a low-level converter might request a font query with a tag of `kIncludeAllFontsBut` and the query result may contain a query with a tag of `kIncludeNoFontsOtherThan`. The value of the tag field returned reflects the results of the query and the meaning of the list of names which follows.

Free virtual memory

The query specified by that tag value `kHintFreeVMTag` and ID value `kHintFreeVMWReclaimId` queries for the amount of free Virtual Memory (virtual memory) in the output device. The result returned is an `SInt32` containing the number of bytes of virtual memory available. If the result is unknown, then -1 is returned.

Miscellaneous

There are couple of additional queries available, but it is highly unlikely that a low-level converter would need these queries or their results. They are given here for completeness.

Spooler Query

The query specified by the tag value `kHintADOSpoolerTag` and ID value `kHintADOSpoolerId` queries for the presence of a spooler. The returned result is of type `TriState`. If the value is `kTriTrue` then the output device is a spooler. If the value is `kTriFalse` then the output device is not a spooler. If the returned value is `kTriUnknown`, then it is unknown whether the job is targeted to a spooler.

PostScript Version Query

The query specified by the tag value `kHintPSVersionTag` with ID value `kHintPSVersionId` queries for the PostScript language version and revision of the output device. The value returned is of type `PSVersion`.

```
struct PSVersion{
 /// revision, -1 => unknown
 long revision;
 /// "\p" => unknown
 Str63 version;
};
typedef struct PSVersion PSVersion;
```

The revision field is a long containing the PostScript revision number of the target output device. This is the number normally returned by the PostScript revision operator. A value of - 1 means unknown.

The version field is a Pascal string containing the PostScript version information as returned by the PostScript version operator. A typical version string is something like "\p2013.106". If the version string is unknown, the length of the string is zero.

It is very unlikely that a generator of PostScript code would request or use the results of the version query. This query is

usually used to generate information for a user, although in rare circumstances it can be useful. More typically, the PostScript language level query is used instead.

PostScript Product Query

The query specified by the tag value `kHintProductTag` and ID value `kHintProductId` queries for the printer product string. The returned result is a Pascal string. The length of data returned in this hint is variable size. If the results for the query are unknown, then a zero length string is returned.

It is very unlikely that a generator of PostScript code would request or use the results of the product query. This query is usually used to generate information for a user.

Back to top

Appendix C: Useful FeatureUtilsLib Routines

The Download Manager and its clients prepare a hints collection for use with each download. This hints collection contains information about the feature requests for that download job. For example, these features can include number of copies and cover page, as well as printer specific features such as duplex, image enhancement, paper tray selection and so forth.

Generators of PostScript output, such as Download Manager low-level converters, know how to generate the device independent PostScript code to image a document, but typically know little or nothing about printer features and how to invoke them. To simplify the task of handling printer specific features, the shared library `FeatureUtilsLib` contained in `PrintingLib`, versions 8.6 and later, was born.

`FeatureUtilsLib` can take the hints collection which contains the job feature information and generate the PostScript language feature code needed to invoke user requested features. This greatly relieves the burden on those clients who know how to generate device-independent PostScript code but would rather not worry about the printer specific features.

Usually there are specific points in the PostScript stream where various printer feature invocations must occur both from the point of view of Document Structuring Conventions (DSC) conformity as well as PostScript execution. For example, if a specific paper tray is used on the first page of a document, the PostScript invocation code of that paper tray must appear outside any page level save/restore nesting on that page, or else the output will be incorrect. Because only the generator of the PostScript page description knows where it is in the process of generating the output stream, that generator must work closely with the `FeatureUtilsLib` code generation to ensure that the correct feature requests are emitted at the proper point in the PostScript output stream.

Generating Feature Code with psWriteSubsectionFeature

The `FeatureUtilsLib` routine `psWriteSubsectionFeature` makes the generation of feature code straightforward. This routine relies on the concept of a `PSSubsection` as introduced in the Download Manager documentation as well as in this document. A `PSSubsection` is a way of communicating DSC and other structural information. `psWriteSubsectionFeature` knows the points it needs to generate the pieces of feature code invocation data; it just needs to be notified by the caller where the caller is in its generation of output.

```
OSStatus psWriteSubsectionFeature(StreamInfoData comm,
   Collection hints,
   PSSubsection subsection,
   void *info,
   psSubsectionLocation subsectionLocation,
   Boolean isNotEPSOutput);

typedef enum psSubsectionLocation{
 kBeforeSubsection = false,
 kAfterSubsection = true
}psSubsectionLocation;
```

- `comm` represents the stream any generated PostScript code is emitted into. The `StreamInfoData` type is described in Appendix A.
- `hints` is a collection representing the job being processed. This collection contains information that `psWriteSubsectionFeature` uses to generate the proper feature code.
- `subsection` is the `PSSubsection` corresponding to the position in the output stream that the caller is either about to write or just wrote.
- `info` is a pointer to a structure relevant to the subsection being written. The value may be `NULL`; otherwise, it will be the data type assigned to the `PSSubsection` corresponding to subsection. See the header file PSStreamInfo.h for the list of `PSSubsection` values and the proper info data type for each `PSSubsection`.
- `subsectionLocation` is either `kBeforeSubsection` or `kAfterSubsection` depending on whether this call to `psWriteSubsectionFeature` is being made before the caller has written the data corresponding to this subsection or after.

`isNotEPSOutput` is a Boolean indicating whether the caller is generating EPS data. Some of the invocation code normally generated by `psWriteSubsectionFeature` is not appropriate when the caller is generating EPS output. If `isNotEPSOutput` is true, then the caller is not generating EPS data and, if it is false, the caller is generating EPS data.

A simple example clarifies this:

```
#include "PSStreams.h"
#include "FeatureUtilsLib.h"

#define DSC30Version 0x30000  // Fixed(3.0);

OSStatus doPercentBang(StreamInfoData comm,Collection hints){

 OSStatus err = noErr;
 Fixed dscVersion = DSC30Version;
 Boolean isNotEPS = true;
 SubsectionStr psVersion = {"\p%!PS-Adobe-3.0\r",
        kSubPSAdobe};

 /*  we are about to write the '%!PS-Adobe-3.0' comment
  beginning our PostScript generation so we first call
  psWriteSubsectionFeature indicating this so that it can
  generate any feature code that must appear before this    comment.
 */
 err = psWriteSubsectionFeature(comm, hints,
    kSubPSAdobe, &dscVersion,
    kBeforeSubsection,  // BEFORE
    isNotEPS);
 /*  Now we emit '%!PS-Adobe-3.0' into the stream. */
 if(!err)err = psOutFormatPositionInfo(comm, &psVersion,
    &dscVersion);

 /*  Now tell psWriteSubsectionFeature that we just wrote
  the '%!PS-Adobe-3.0' comment. */
 if(!err)err = psWriteSubsectionFeature(comm, hints,
     kSubPSAdobe, &dscVersion,
    kAfterSubsection,  // AFTER
    isNotEPS);

 return err;

}
```

By using this stylized way of emitting PostScript output code into the output job stream, the sample code automatically gets a cover page before the job, should the hints collection indicate that it is required. When the code sample calls `psWriteSubsectionFeature` before the initial `%!PS-Adobe-3.0` emitted into the print stream and indicates that it is about to write the subsection `kSubPSAdobe`, the routine examines the supplied hints collection to see if it indicates that a cover page should be generated before the job. If a cover page should be generated, then `psWriteSubsectionFeature` writes it into the output stream and ends the cover page job so that when the above code fragment then emits `%!PS-Adobe-3.0` into the output stream, that is the first PostScript code appearing in the print job following the cover page. Note that if `isNotEPS` is false, `psWriteSubsectionFeature` does not generate a cover page.

To use `psWriteSubsectionFeature` properly, the caller must carefully identify the different parts of the PostScript output that it generates. This also encourages the generators of PostScript code to follow the DSC guidelines to emit structured PostScript code. The sample code supplied with this document follows this approach and forms a good basis for starting any Download Manager converter module.

Detecting Manual Feed

A low-level converter needs to be able to respond to the call `psLowGetStreamInfo` to inform the caller whether a given print job requires manual feed. The `FeatureUtilsLib` routine `psRequiresManualFeed` is available to aid a low-level converter's efforts to respond. A low-level converter that uses the `psWriteSubsectionFeature` described above to handle its feature code should use `psRequiresManualFeed` to determine if the print job requires manual feed.

```
OSStatus psRequiresManualFeed(Collection jobHints,
   Boolean *requiresManualFeedP);
```

- `jobHints` is a Collection corresponding to the job collection for the download job in question.
- `requiresManualFeedP` is a pointer to a Boolean which is filled in by the call. If the `jobHints` collection indicates that the job requires manual feed `*requiresManualFeedP` is set to true; otherwise, it is set to false.

Note:
`psRequiresManualFeed` does not take into account whether an EPS job is being generated. Because of this, it is important that a low-level converter determine whether EPS output is to be generated and, if so, it needn't bother calling `psRequiresManualFeed` but instead can simply return that the job does not require manual feed. The sample converter properly handles this situation.

Back to top

Downloadables

	Acrobat version of this Note (K).	Download
	Binhexed Sample Code (343K).	Download

Back to top