Technote 1144

Writing Custom Hoses For LaserWriter 8.6

by Richard Blanchard and Ingrid Kelly
Apple Worldwide Developer Technical Support

CONTENTS

LaserWriter 8.6 supports printing to a variety of desktop printer types, including PAP, LPR, and IrDA. Each desktop printer can have its own method of communicating with its associated physical printer, RIP, or other post-printing processor. LaserWriter 8.6, through the invention of custom "hoses", adds the ability for shared libraries to implement various communication methods and for these libraries to be loaded dynamically based upon a DTP's type. This Technote gives an overview of the custom hose specification for developers.

Identifying a DTP's Type

Associated with each LaserWriter 8 desktop printer is a `'PAPA'` resource that identifies the DTP's type, as well as the printer's name and other associated communications parameters. The DTP type takes the form of a four-byte constant. All DTP types, with the exception of the `'PAP '` type, must be in the form `'=XXX'` (e.g. `'=Hld'` or `'=Fil'`). This is due to how the old `'PAPA'` resource was expanded with the release of the LaserWriter 8.5.1 driver. The new DTP type signature replaces the zone string in the old 103-byte `'PAPA'` resource and maps to the appropriate tag `'TYPE'` in the `'PAPA'` resource. Technote 1115: The Extended 'PAPA' Resource contains detailed information about the contents of the `'PAPA'` resource.

LaserWriter 8.6 DTP Types

The `PrintingLib` that ships with LaserWriter 8.6 supports the following DTP types:

'PAP '
Communication with the printer is performed using AppleTalk's Printer Access Protocol (PAP). The printer's AppleTalk name, type, and zone are stored in the compatibility portion of the `'PAPA'` resource as described in Technote 1115: The Extended 'PAPA' Resource.

'=Hld'
The hold desktop printer is unique in that there is no associated communications module for it. A hold desktop printer never converts the desktop spool file to PostScript, but instead simply queues the spool files. To print a spool file queued to a hold desktop printer, the user must move the spool file into the queue of another type of desktop printer.

'=Fil'
A translator desktop printer writes its PostScript, EPS, or PDF output to a file.

Note:
The ability to output PDF is dependent upon the user having Adobe's Acrobat Distiller installed.

'=LPR'
An LPR desktop printer uses the Unix LPD protocol to communicate over TCP/IP with a print server. For more information on this protocol, please see *RFC 1179* at <http://ds.internic.net/ds/dspg1intdoc.html>.

'=Cst'
The custom application desktop printer writes its PostScript to disk and then launches an application to post-process the PostScript job. See Technote 1113: Customizing the Desktop Printer Utility for additional information on custom DTPs.

'=Ird'
The PostScript job is transmitted using an infrared link to an IrDA-capable printer. LaserWriter 8 uses the IrDA specification as outlined at http://www.irda.org.

Note:
Future releases of the LaserWriter 8 driver may add built-in support for USB and FireWire.

Adding Hose Plug-ins

When printing a job to a desktop printer, LaserWriter 8.6 obtains the four-byte DTP type and then looks for a shared library containing the matching hose. LaserWriter 8.6 searches for the library in the following order:

1. In the System's "Printing Plug-ins" folder (in the Extensions folder)
2. In the `PrintingLib` file

A desktop printer's type is obtained using the `SettingsLib` call `psGetDTPType()`, as described in Technote 1129: The Settings Library. The Printing Plug-in Manager (as described in an upcoming Technote) is then used to find a printing plug-in of type 'hose' with a subtype matching the DTP type.

Plug-in files managed by the Printing Plug-ins Manager, such as custom hoses, are required to have a resource of type `'PLGN'` with ID -8192 that contains information about the plug-ins contained in a given file. If they do not, they cannot be used and are ignored by the LaserWriter 8.6 driver. The plug-ins are also required to have a standard `'cfrg'` resource describing the code fragments in the data fork of the file.

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 library handles, and the library name.

The `'PLGN'` resource is as follows:

```
 short num;                // the number of shared libraries
 PluginLibInfo libInfo[num];

The PluginLibInfo structure is as follows:
 typedef struct PluginLibInfo{
  SettingsDataType type;
  SettingsDataSubType subtype;
  unsigned char libraryName[];  // pascal string
                                // word aligned
 }PluginLibInfo;
```

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

A ResEdit `'TMPL'` for editing the `'PLGN'` resource is provided in PrintingLib 8.6. The type field should be `'hose'` and the subtype should be `'=XXX'`, where XXX is the custom-registered type of DTP being supported. See the Hose Type Registrations section for more details on registration. The `libraryName` should match the name of the code fragment in the data fork which implements the hose.

The Hose Interface

hoseOpen

Hose fragments are required to export a single entry point. This entry point, `hoseOpen()` has the following signature:

OSStatus hoseOpen(HoseInfo *hoseInfo, const BufCallbacks *callbacks, Collection hints, Handle papaH);

The hoseOpen routine's primary job is to fill out the structure pointed to by hoseInfo.

/* The HoseInfo structure is filled out by a `hoseOpen` procedure. The
structure describes the buffer requirements of the hose as well as
the function pointers for reading, writing, and closing the hose.

```
*/
typedef struct{
 HoseOutProc out;        // Called to write a buffer.
 HoseInProc in;          // Called to read a buffer.
 HoseIdleProc idle;      // Called periodically.
 HoseCloseProc close;    // Called to shut down the connection.
 HoseConnProc connState; // This procedure returns the state of the
                         // current connection.
 HoseStatusProc status;  // Return the hose's current status string.
 HoseDisposeProc dispose;// The hose should free up all of its memory.
 Size bufSize;           // The size of each allocated data buffer.
 long minBufs;           // The hose requires this many buffers. If
                         // there isn't enough memory to allocate them,
                         // the client will return memFullErr.
 long maxBufs;           // We'll never allocate more buffers than this.
 void *refcon;           // A pointer that will be passed to the hose
                         // routines.
}HoseInfo;
```

First, the hose needs to fill in the `out`, `in`, `idle`, `close`, `connState`, `status`, and `dispose` fields with native function pointers to routines in the hose that implement the hose's functionality (i.e., the PPC code provides PPC function pointers while the 68KCFM code provides 68KCFM function pointers. No classic 68K function pointers are supported). In addition to filling in the hose's function pointers, the hose must fill in the `bufSize`, `minBufs`, and `maxBufs` fields to describe the hose's buffer requirements. Lastly, the hose needs to fill in the `refcon` field with a pointer to its own storage.

Before calling the hose to transmit data, the hose client provides buffering to improve performance. Because of this buffering, the hose must deal with only one transmit buffer and one receive buffer at a time. The hose client is responsible for allocating the buffers that it uses for providing the buffering. The hose indicates the size of the buffers its client should allocate by filling in the `bufSize` field during `hoseOpen`. Some typical buffer sizes include 4096 bytes for the `'PAP '` hose and 16384 bytes for the

'=Fil' hose. The hose specifies the minimum number of buffers it needs in minBufs. It is recommended that the minBufs field be set to at least four for bidirectional hoses and to at least two for unidirectional hoses. The maximum number of buffers to be allocated is set by the hose in the maxBufs field. The current rule of thumb for this field is that it should not specify more than 256K worth of memory. In other words, it should be (262144 / bufSize). This is just a general rule of thumb; some hoses may require a larger maxBufs setting, while others will use a smaller number.

The callbacks parameter to hoseOpen is a pointer to a BufCallbacks structure filled in by the hose client.

```
typedef void (*FinishedWriteProc)(MemQElemPtr memElem, OSStatus err);
typedef void (*FinishedReadProc)(MemQElemPtr memElem, OSStatus err);

typedef struct{
 FinishedWriteProc finishedWrite;
 FinishedReadProc finishedRead;
}BufCallbacks;
```

The hose's client places two pointers to native functions in this structure. These functions are to be called by the hose when a read or write is completed. For more information on these function pointers, see the detailed description of HoseOutProc and HoseInProc below. The hose should store a copy of this structure in its private data.

As part of the hoseOpen call, the hose is handed a Collection Manager collection containing hints for configuring the job that will be transmitted through the hose. The hints in this collection are used by the QuickDraw to PostScript converter hose client to configure the byte codes that are available for its generation of PostScript language output. In particular, the kHintEighthBitTag and kHintTransparentChannelTag hints, as defined by LaserWriter 8.6, configure the hose client as to whether it can generate its data using byte values outside the standard PostScript language printable ASCII character set. These hints are defined as follows:

```
/* When generating PostScript for the output stream, the PostScript converter
 will by default use, if needed, characters in the range 0x80-0xFF
 inclusive. Use the 'kHintEighthBitTag' with a value of 'false' to
 prevent the converter from emitting bytes with the high
 bit set.
*/
#define kHintEighthBitTag   'bit8'
#define kHintEighthBitId   1
#define kHintEighthBitVariableType  Boolean
#define kHintEighthBitDefault  true

/* When generating PostScript for the output stream, the PostScript converter
 will by default use, if needed, characters in the range 0x00-0x1F
 inclusive. Use the 'kHintTransparentChannelTag' with a value of
 'false' to prevent the converter from emitting bytes less than 0x20.
*/
#define kHintTransparentChannelTag  'trns'
#define kHintTransparentChannelId   1
#define kHintTransparentChannelVariableType Boolean
#define kHintTransparentChannelDefault  true
```

Typically, the hints collection passed to hoseOpen does not contain the collection items corresponding to these tag/id pairs. This is equivalent to the kHintEighthBitTag and kHintTransparentChannelTag hints both set to true, i.e. bytes 0x00-0xFF are all available.

For hoses which communicate through a channel that has attributes more restrictive than these defaults, the hose must add the appropriate collection item(s) to the hints collection to ensure that the hose client

only writes bytes in the supported range. Note that only hints which are *more* restrictive than the defaults need to be added. If a hose supports full 8-bit communications, it need not add these hints to the hints collection passed in.

In some cases, the hints collection passed to `hoseOpen` already contains either or both of the `kHintEighthBitTag` and `kHintTransparentChannelTag` collection items and the collection item may be locked. If the hose client requires a more restrictive setting than that present, it *must* add the hint to the collection, regardless of whether the hint is already locked.

```
OSErr err;

// e.g. this hose cannot transmit the high 8 bit
kHintEighthBitVariableType eightBit = false;

// unlock the hint if it is already there
// this is OK if the hint is already unlocked
err = SetCollectionItemInfo(hints, kHintEighthBitTag,
   kHintEighthBitId, collectionLockMask,
   0);
// if the hint isn't already there that's fine
if(err == collectionItemNotFoundErr)
  err = noErr;

if(!err){
 err = AddCollectionItem(hints, kHintEighthBitTag,
    kHintEighthBitId, sizeof(eightBit),
   &eightBit);
 if(!err)
  err = SetCollectionItemInfo(hints,
    kHintEighthBitTag, kHintEighthBitId,
   collectionLockMask, collectionLockMask);
}
```

See *Inside Macintosh* : QuickDraw GX Environment and Utilities for more information on the Collection Manager.

The last parameter to `hoseOpen`, `papa`, is the handle to an extended `'PAPA'` structure. Please see Technote 1115: The Extended 'PAPA' Resource for a full description of this structure and Technote 1129: The Settings Library for the `'PAPA'` accessor routines, `psPapaToCollection` and `psCollectionToPapa`. Each type of hose has different communications parameters that specify the target output device and how the communications channel is to be configured. The `'PAPA'` handle provides those communications settings for the hose and are set by the creator of the desktop printer.

In its `hoseOpen` routine, the hose should allocate any needed memory and begin opening the connection with the printer. The hose need not -- and for most connection types should not -- complete the connection in the `hoseOpen` call. Because of the lengthy connect times of most communications techniques, the opening of the printer connection should take place asynchronously. The hose starts the connection process and then returns `noErr` from `hoseOpen`. The client will periodically call the `hoseConnProc` requesting the current state of the connection. The `hoseConnProc` looks like this:

```
typedef enum{
 kConnClosed = 0,  // Start in this state.
 kConnOpening,     // This is the state while we wait for the
                   // printer to accept the connection.
 kConnOpen,        // This is the state while we do reads and
                   // writes to the printer.
 kConnClosing      // This is the state while we wait for the
```

```
                    // connection to close.
}ConnState;

typedef ConnState (*HoseConnProc)(void *refcon);
```

The hose's connection procedure should return the constant `kConnOpening` while the connection is being established. Once the open completes successfully, calls to the hose's connection procedure should return `kConnOpen`. If instead an error occurs while opening the hose, the hose's connection procedure should return `kConnClosed`. At that point, the hose client will call the hose's `HoseCloseProc` and `HoseCloseProc` needs to return the appropriate error code indicating why the hose couldn't be opened.

HoseOutProc
The primary purpose of a hose is to transmit data. This is accomplished by the hose's client through calls to `HoseOutProc`. `HoseOutProc` looks like this:

typedef OSStatus (*HoseOutProc)(void *refcon, MemQElem *memElem);

and thus the hose's function is:

OSStatus hoseOut(void *refcon, MemQElem *memElem);

The `refcon` parameter passed to `HoseOutProc` is taken from the refcon field of the `HoseInfo` structure filled out by the `hoseOpen` routine. This value should be a pointer or a handle to the hose's private data.

The second parameter to `HoseOutProc`, `memElem`, is a pointer to a `MemQElem` structure describing the data to be written.

```
typedef struct MemQElem{
 QElemPtr qLink;      // Used by Enqueue and Dequeue- private.
 short qType;         // Our constant (kMemQueueType) to identify our
                      // queues- private.
 struct BufIO *bufIO; // So we can recover buffer information-
                      // private.
 Byte *buf;           // Pointer to the allocated buffer.
 SInt32 maxBytes;     // The size of the block pointed to by 'buf'
 SInt32 nBytes;       // Number of valid bytes in 'buf'.
 Boolean eoj;         // true if the data is followed by an end of
                      // job.
 Boolean inQOnly;     // This buffer should be used only for the input
                      // routines- private.
}MemQElem, *MemQElemPtr;
```

Note:
Many of the fields of the `MemQElem` structure are private and are used by the hose client. These fields are marked private and must not be used by the hose.

The hose client uses the `'eoj'` field of the `MemQElem` structure to signal to the hose when it needs to transmit a PostScript end of job to the printer. If the `'eoj'` field of the `MemQElem` structure is true, a PostScript end of job indication must be sent after or along with the buffer of data (if any) in this `memElem`. For some communications channels, this `'eoj'` is a data byte sent after the data, such as control-D for serial connections. For other communications channels, such as `PAP`, the end of job indicator is out of band with the data itself.

Again, it is highly recommended that hoses perform their writes and reads in an asynchronous manner. In

this case, HoseOutProc begins to write the nBytes pointed to by buf and then returns to the caller. When the write completes, the hose signals the caller by calling the finishedWrite function pointer passed in the BufCallbacks structure to hoseOpen.

typedef void (*FinishedWriteProc)(MemQElem *memElem, OSStatus err);

When making the FinishedWriteProc call, the hose passes in the MemQElem pointer passed to hoseOut along with an error code. If the write finished successfully, the error code should be noErr. If there was an error in the write, pass that code to FinishedWriteProc.

The call to FinishedWriteProc is an indication from the hose to the client that the hose is done with the MemQElem structure and is ready for another hoseOut call. In fact, the routine called through the FinishedWriteProc may immediately make another hoseOut call before returning to the hose. Because of this, the hose must be prepared for the hoseOut routine to be invoked while still in an asynchronous completion routine. Furthermore, once the MemQElem pointer is passed to FinishedWriteProc, the hose should no longer reference it. Any data that might have been copied from the structure before calling FinishedWriteProc is no longer valid (particularly 'buf'). So, not only is the structure itself no longer valid, any data contained in the structure that was previously in use is also no longer valid.

HoseInProc

If a hose is managing a unidirectional communications channel, the hose need not have a routine for reading data. In this case, the hoseOpen routine should fill in the 'in' field of the HoseInfo structure with NULL during hoseOpen.

If the hose can read data from the printer, it fills the in field of the HoseInfo structure with a pointer to its routine that reads data.

typedef OSStatus (*HoseInProc)(void *refcon, MemQElem memElem);

As with HoseOutProc, HoseInProc should execute asynchronously. When invoked, HoseInProc should start a read. When data is available and the hose's asynchronous completion routine is invoked, the hose invokes the client's FinishedReadProc, passing back the MemQElem pointer passed to HoseInProc and an error code. The hose cannot read more than memElem->maxBytes bytes. In addition, the hose must fill in memElem->nBytes with the number of bytes read into memElem->buf.

typedef void (*FinishedReadProc)(MemQElem memElem, OSStatus err);

As with data writes, once the FinishedReadProc is called, the hose must be prepared for another call to HoseInProc before FinishedReadProc returns. Similar to FinishedWriteProc, the data in the MemQElem should be considered invalid after the FinishedReadProc is called by the hose.

HoseIdleProc

Not all hoses are able to use asynchronous completion routines to note the end of a read or write. To help these hoses, a HoseIdleProc can be specified in the HoseInfo structure returned from hoseOpen. If the idle field of the HoseInfo structure is not NULL, the hose's client will periodically call the HoseIdleProc. This idle procedure can check the status of pending reads and writes and call FinishedWriteProc and FinishedReadProc as needed. Most hoses do not need a HoseIdleProc, but, if one is needed, it has the following signature:

typedef OSStatus (*HoseIdleProc)(void *refcon);

HoseStatusProc

While a hose is open, the hose's client may periodically request that the hose query the printer for status. When the call is made the hose should copy a Pascal string describing the printer's last known status into the buffer pointed to by statusStr. The hose should also start an asyncronous status request to the printer. When the asyncronous status request returns the hose must hold that status until the next call to its status procedure.

```
typedef OSStatus (*HoseStatusProc)(void *refcon, StringPtr statusStr);
```

Note:
Make sure that the `StringPtr` always points to a valid Pascal string (i.e., leave the length byte 0 until the whole string is written so that your client does not display garbage characters if the timing is wrong).

For some communications channels, such as serial channels, the status from a printer is returned on the back channel read by the `HoseInProc`. In such a case, the hose's client pulls the status out of the back channel and the hose does not need to do anything other than transmit a status request to the output device.

HoseCloseProc

When the client is done with the hose, it calls the hose's `HoseCloseProc` to shut down the connection. The hose should terminate any pending reads and writes and begin to shut down the connection. If this shut down procedure is immediate, it can be completed before this routine returns. If the shut down procedure takes an extended amount of time, this routine can begin the process and return.

```
typedef OSStatus (*HoseCloseProc)(void *refcon);
```

HoseDisposeProc

After `HoseCloseProc` is invoked, the `HoseConnProc` is called repeatedly until `kConnClosed` is returned. When the hose signals that the connection has been shut down, `HoseDisposeProc` is called to allow the hose to release any memory it still holds.

```
typedef OSStatus (*HoseDisposeProc)(void *refcon);
```

If there was an error during an asynchronous close, the `HoseDisposeProc` should return a non-zero error value.

Hose Type Registrations

In order to ensure that we do not have conflicting hose types, we ask that you register your custom hose 4-byte type by sending an email to devprograms@apple.com. Please send the following information to register your custom hose type:

1. Contact Name
2. Company Name
3. Mailing Address
4. Phone Number
5. Email Address
6. Make and Model of device
7. Description of communications method
8. 4-byte type (in the form '=XXX')

Summary

As outlined in this document, creating a custom hose for LaserWriter 8.6 is fairly straightforward and clean. Make sure that you are familiar with the other documents mentioned in this Technote before you begin on your journey to create a custom hose!

References

Technote 1113: Customizing the Desktop Printer Utility

Technote 1115: The Extended 'PAPA' Resource
Technote 1129: The Settings Library
Technote 11XX: Printing Plug-ins Manager Specification (coming soon)
Inside Macintosh : QuickDraw GX Environment and Utilities

Downloadables

 Acrobat version of this Note (K).

 Binhexed Routine Descriptor Lib (9K).

Acknowledgments

Thanks to John Blanchard, David Gelphman, and Dave Polaschek.

To contact us, please use the Contact Us page.
Updated: 2-November-98