Technical Note #5139

01 November 1993

Adobe Systems Incorporated
Adobe Developer Technologies
345 Park Avenue
San Jose, CA 95110
http://partners.adobe.com/
Contents

**PSPrinter's PrGeneral Interface to Get Communications Resources** 5

1 The New PRGeneral Opcodes 5
   Using kLoadCommProcsOp and kUnloadCommProcsOp 6

2 Filling CommRecord Fields, idleProc and idleRefCon 8

3 Using the Communications Channel 9
   Initiating Communication 9
   Sending Data Down the Communications Channel 10
   Reading Data From the Printer 11
   Indicating the End of Job to the Printer 12
   Getting Status from the Printer 12
   Terminating the Connection 12

4 Pseudocode of a Typical Job 12

**APPENDIX A** 15

A.1 PRGeneral_Comm.c File Contents 15
A.2 PRGeneral_Comm.h File Contents 23

**Index** 27
PSPrinter’s PrGeneral Interface to Get Communications Resources

The first release of the Adobe™ PostScript™ printer driver for the Macintosh® is called PSPrinter 8.0. This driver is also distributed by Apple Computer as LaserWriter® 8.0 and by Hewlett Packard Company as LaserJet® 8.0. Starting with PSPrinter version 8.1, applications can access PSPrinter’s communications code resources. These resources provide the means to perform very specific low-level tasks such as opening and closing a connection to the printer, beginning and ending a PostScript language job, reading and writing data to the printer, and obtaining status from the printer.

This document describes how an application can access PSPrinter’s communications code resources through the Macintosh Print Manager’s PrGeneral call.

Note For more information on PrGeneral, please refer to Apple’s Inside Macintosh Volume V, pages 410–419.

1 The New PRGeneral Opcodes

Two new PrGeneral opcodes, kLoadCommProcsOp and kUnloadCommProcsOp, have been added to PSPrinter 8.1 and will be supported in subsequent releases of the printer driver including Adobe’s GX driver. For an application to know if these opcodes are available, it could find out the driver version by checking the value returned by PrDrvrVers(). For versions 8.1 and 8.1.1, the value 81 (decimal) is returned. If the code resources are not available, the PrGeneral calls with opcodes kLoadCommProcsOp and kUnloadCommProcsOp will return errors.

Note There is a bug in PSPrinter versions 8.1 and 8.1.1, for which we have provided a work-around. Please refer to the code in Appendix A, page 13. This bug has been fixed in PSPrinter version 8.1.2 and thus this work-around should not be executed with 8.1.2 and subsequent versions of the driver. For version 8.1.2, the value 82 (decimal) is returned by PrDrvrVers().

Appendix A details the data structures proposed for use in application development. These data structures are referred to throughout this document. Please refer to Appendix A as appropriate.
1.1 Using **kLoadCommProcOp** and **kUnloadCommProcOp**

To use the new PrGeneral **kLoadCommProcOp** opcode, fill a **TLoadCommProc** structure with an **iOpCode** of **kLoadCommProc** and a **version** of value **kCommProcsVersion**. This specifies what version of **kLoadCommProc** the application is expecting. On exit, the driver checks to see if this value is greater than its version. If it is, the driver puts its version here to inform the application of this lesser version. The final version is thus the minimum of the version specified by the application and that returned by the driver. The application should check this value to see if it has been changed by the driver. If the value is not what is expected, the application must decide whether or not it can use the present driver version. There is currently only one possible value for **kCommProcsVersion**.

The **commRecP** field of the **TLoadCommProc** structure must be filled with a pointer to a structure that is at least as large as a **CommRecord** structure. The caller may allocate a structure that begins with a **CommRecord** structure, followed by private fields to be used as needed by the application. The **whichComm** field of the **TLoadCommProc** structure must be filled with a **CommSelector** constant. Currently two **CommSelector** constants are defined. The first **CommSelector** constant, **kCommDefault**, is used most often and asks the printer driver to open its default communications channel. In PSPrinter version 8.1, this default communications channel is an AppleTalk® PAP connection. The second **CommSelector** constant, **kCommFile**, causes the printer driver to open a communications channel that saves PostScript language output to disk.

The following example shows how to load a communications module using the PrGeneral call:

**Example 1: Loading the Communications module**

```c
CommRecord *commPtr;
void LoadComm()
{
    TLoadCommProc loadCommProc;

    commPtr = (CommRecord *)NewPtr(sizeof(CommRecord));
    if (commPtr){
        loadCommProc.version = kCommProcsVersion;
        loadCommProc.whichComm = kCommDefault;
        loadCommProc.commRecP = commPtr;
        loadCommProc.ignoreBGflag = false;
        PrGeneral((Ptr)&loadCommProc);
        if (loadCommProc.iError == noErr){
            /* Now we can call the comm module */
        }
    }
}
```
Note  The file PRGeneral_Comm.h listed in Appendix A.2 contains macros, such as CommOpen() and CommOut(), for convenient access to the procedure pointers. This document contains prototypes as if these macros were functions. These macros are described fully in Section 3 of this document.

If the ignoreBGflag field of the TLoadCommProcs structure is true, CommOpen() opens communication to the printer, regardless of whether the user has selected background printing or not. If ignoreBGflag is false, the PrGeneral routine checks whether background printing is on, and returns that information in the background field of the TLoadCommProcs structure.

If background printing is on, the driver sets the backChannel flag to false. backChannel is a field of the ChannelCaps structure which is in turn a field of the CommRecord structure. The application should just look at the backChannel flag and not respond to whether background printing is on or off. If backChannel is false, CommOpen() creates a spoolfile, and CommOut() writes the PostScript language code to the spoolfile, just as a normal print job would do. PrintMonitor then prints the job later as usual. When backChannel is false, the driver does not generate any PostScript language code of its own. If backChannel is true, the application can generate queries and read responses, for example, to see which fonts are available in the printer. If backChannel is false, the application should not generate queries and cannot read responses. In this case, the application should assume that there are no built-in fonts.

Note  If the job is printing in background, the PrGeneral call PSpict2eps cannot be used at print time to convert a PICT to EPS. This is an unfortunate side effect of the driver not being reentrant, and both printing in background and PICT to EPS generate print jobs. An alternative to calling PSpict2eps at print time is to convert imported PICT objects to EPS objects, thus calling PSpict2eps at import time rather than at print time. For more information on PSpict2eps, please refer to Technical Note #5134, “Tips for Working with the Adobe PostScript Printer Driver for the Macintosh”, available from the Adobe Developers Association.

Upon loading the communications module, if the kLoadCommProcsOp PrGeneral call completes without an error, the driver has set the flags in the channel field of the CommRecord, and filled the six CommRecord fields: commOpen, commIn, commOut, commEoj, commStatus, and commClose. These fields contain Pascal-style function pointers to the communications module’s six entry points. If a function is not implemented, its function pointer will be NULL (as is the case with commIn when backChannel is false). The remaining CommRecord fields are set to 0’s. The function pointers are valid until the kUnloadCommProcsOp PrGeneral call is made.
The following example shows how to unload a communications module using the PrGeneral call. Note the use of the new PrGeneral opcode kUnloadCommProcsOp.

Example 2: Unloading the Communications module

```c
void UnLoadComm()
{
    TUnloadCommProcs unloadCommProcs;
    unloadCommProcs.iOpCode = kUnloadCommProcsOp;
    unloadCommProcs.version = kCommProcsVersion;
    unloadCommProcs.commRecP = commPtr;
    PrGeneral((Ptr)&unloadCommProcs);
    DisposePtr((Ptr) commPtr); /* assuming NewPtr was successful */
}
```

Filling CommRecord Fields, idleProc and idleRefCon

Once the kLoadCommProcsOp PrGeneral call finishes, the application has the option of filling the idleProc and idleRefCon fields in the CommRecord. If the idleProc field is not NULL then the communications module calls the application’s idle procedure as often as possible. The application’s idle procedure should have the following form:

```c
pascal OSErr appCommIdle(long idleRefCon, Boolean deadTime);
```

As its first parameter, the application’s idle procedure is passed the contents of the CommRecord’s idleRefCon field. The application can use this parameter as a pointer to whatever data may be needed in the idle procedure.

For good throughput it is important that the application’s idle procedure not call WaitNextEvent(), GetNextEvent(), or EventEval() too often. At the same time, the idle procedure must give up some time if other tasks on the Macintosh are to receive time. The second parameter to the idle procedure deadTime is true if the communications module is blocked from sending data to the printer. If deadTime is true then the idle procedure should call WaitNextEvent(), GetNextEvent(), or EventEval(). If deadTime is false then the idle procedure should start a timer and ensure that time is given to background applications at least once a second.

The following example shows a sample application idle procedure.

Example 3: Sample IdleProc

```c
#define kGiveUpTimeInterval30

typedef struct {
    long lastWaitNextEvent;
```
pascal OSErr appCommIdle(long idleRefCon, Boolean deadTime)
{
    IdleData *idleDataP;
    EventRecord theEvent;
    OSErr err = noErr;

    idleDataP = (IdleData *)idleRefCon;
    if(idleDataP){
        if (deadTime || (idleDataP->lastWaitNextEvent +
            kGiveUpTimeInterval < TickCount())) {
            WaitNextEvent(everyEvent, &theEvent, 0L, NULL);
            idleDataP->lastWaitNextEvent = TickCount();
            if (userCanceled(&theEvent)) {
                err = iPrAbort;
            }
        }
    }
    return err;
}

Note In the previous example, userCanceled() is a function that checks to see if
the event is a command-period or another event that interrupts the job. The
driver will not do any command-period testing of its own.

When the application’s idle procedure returns an error, the current call to
CommOpen(), CommOut(), or CommIn() is aborted and these routines pass
back the idle procedure’s error code. The connection is not closed, so the
application must still call CommClose() and unload the communications
module as in Example 2.

3 Using the Communications Channel

3.1 Initiating Communication

pascal OSErr CommOpen(CommRecord* comm, void *openInfo);

The application must call the CommOpen() function to begin using the
communications channel. CommOpen() takes two parameters. The first
parameter is a pointer to the CommRecord that has been filled by the
kLoadCommProcs PrGeneral call. If whichComm is kCommFile, the
application should pass a pointer to a CommLoadInfo structure as the second
parameter. This structure specifies the location, creator, and type of the file to
be created. If whichComm is kCommDefault and background is true, the
application should pass a pointer to a bgInfoRecord structure as the second
parameter. This structure provides more information for PrintMonitor to
display in its status field and is described in Appendix A. If whichComm is
kCommDefault and background is false, the second parameter can be NULL
since the driver ignores this argument in this case.
CommOpen() returns immediately, even though opening a PAP communication takes several seconds. The first time CommOut() needs to send data, it waits for the connection to open, calling the application supplied idle procedure with the dead time equal to true (refer to Section 3.2 below).

When CommOpen() returns, the ChannelCaps structure in the CommRecord will have been filled with data describing the characteristics of the communications channel.

### 3.2 Sending Data Down the Communications Channel

```pascal
OSErr CommOut(CommRecord* comm, void *buffer, long numBytes);
```

Once the communications channel has been opened, the application can send data down the channel using the commOut field in the CommRecord. The CommOut() function takes as its first parameter a pointer to a CommRecord that has been filled by CommOpen(). The second parameter, buffer, is a pointer to the data to be sent down the communications channel. The third parameter, numBytes, specifies the number of bytes to be sent from buffer. CommOut() places the caller’s data into the communications module’s internal buffers. These data are sent down the connection when the buffers become full or when the write buffers are flushed explicitly by the caller. To cause the communications module to flush its buffers to the channel, pass NULL as the buffer parameter and set numBytes to the constant kFlushWriteBuffers. If background is true, a flush also signals a new page. This enables PrintMonitor to display the number of pages remaining.

The data that an application writes to a communications channel using CommOut() must conform to the channel’s characteristics as described in the ChannelCaps structure in the CommRecord. Specifically, the application must abide by the transparent and eightBit fields in the ChannelCaps structure. If the transparent flag is true, then the application may send characters in the range: 0 – 31, inclusive, over the channel without these characters being treated as communications channel control characters. If transparent is false then the application should not send characters in the range: 0 – 31 inclusive, as data. When transparent is false, a character in this range may interfere with the operation of the channel, as is the case with a Control–D on a serial connection; or the communications channel may remove or change characters in the range 0 – 31, as is the case with the conversion from carriage-returns to line-feeds. The eightBit flag indicates that the channel can carry characters in the range: 128 – 255, inclusive. If eightBit is false then the application should not send characters in the range: 128 – 255, inclusive.
Note Adobe PostScript interpreters treat all white-space characters the same so it is acceptable to send white space characters within the PostScript language code being sent down to the printer. For more information on white-space characters and the PostScript language, please refer to the PostScript Language Reference Manual, pages 26 – 27.

3.3 Reading Data From the Printer

    pascal OSERROR CommIn(CommRecord* comm, void *buffer,
                          long *numBytes, Boolean *eoj);

Data can be read from the printer via the communications channel if the backChannel flag in the CommRecord’s ChannelCap structure is true. When calling CommIn(), the application should pass in the CommRecord filled by CommOpen(). The buffer parameter points to a block of memory that is filled by CommIn(). On entry to CommIn(), *numBytes is the number of bytes the caller would like to read from the communications module’s buffers. On exit from CommIn(), *numBytes contains the number of bytes actually read into buffer. The communications module only returns data that are immediately available. If the application sends down a query with one PostScript language flush at the end, it can require several calls to CommIn() to read the responses. It is up to the application to know if more data are expected. If on entry, *numBytes is greater than zero but there are no bytes available in the communications module’s buffers, then CommIn() returns immediately with *numBytes equal to zero.

Note Some devices do not automatically flush buffers upon receipt of a PostScript language flush. As a result the application must know how much data to expect back and be prepared to either keep reading or stop reading responses from the device.

The application usually calls CommIn() during the print job to look for PostScript language errors and other status messages, such as out of paper. The communications module does not do any error checking of its own.

If the printer on the other end of the communications channel sends an end of job indication (EOJ) in response to an EOJ sent by the application (refer to Section 3.4 below), then it is reported in the *eoj boolean. The *eoj flag can be returned as true at any time and if *numBytes is returned as non-zero then the EOJ was received after the bytes in *buffer. If the ChannelCap structure’s eojHandshake flag is true then after sending an EOJ signal, the application must call CommIn() repeatedly until either an error is encountered or the communications channel reports an EOJ was received.

The application may want to know the number of bytes available in the communications module’s buffers without actually reading them. To do this, pass in NULL for the buffer parameter. If buffer is NULL then on exit *numBytes contains the number of bytes available.
3.4 Indicating the End of Job to the Printer

```
pascal OSErr CommEoj(CommRecord* comm);
```

Use the `CommEoj()` function to send an end of job (EOJ) indication over the communications channel. The application must wait to receive EOJ if the `ejHandshake` field of the `ChannelCaps` structure is `true`, otherwise it is optional. After sending the complete job, the application can wait for the receiving EOJ to look for PostScript language errors, for example.

3.5 Getting Status from the Printer

```
pascal OSErr CommStatus(CommRecord* comm);
```

Use the `CommStatus()` function to request that the communications module get the current channel status at the next convenient moment. The status field of the `CommRecord` contains a Pascal string that is the channel’s status. This status is not updated on exit from `CommStatus()` but rather is updated asynchronously by the communications module.

**Note** There is no definitive way to know if the status from the printer has changed. There is therefore no definitive way to tell when the application should update the status dialog, except by comparing the new string value with the previous string value.

3.6 Terminating the Connection

```
pascal OSErr CommClose(CommRecord* comm);
```

Call `CommClose()` to shut down the connection on the communications channel. If the application is completely done with the communication it can unload the communications module as in Example 2 and call `PrClose()`. The application should not close the connection after doing a job query. In this case, it is enough to send an EOJ, wait to receive an EOJ and then continue with the print job.

4 Pseudocode of a Typical Job

A typical job has the following pseudocode components:

```
/* Initiate communication */
PrOpen();
LoadComm(); // app supplied routine to load the comm module

/* Query the printer: only in foreground */
CommOpen();
CommOut(); // send the query
CommIn(); // read printer query response
CommEoj(); // read EOJ
CommIn(); // read printer EOJ response
```
/* Send PostScript language job */
CommOut(); // send some PostScript language data
CommIn(); // check backchannel for PostScript language errors
.
.  // keep calling CommIn() until you have received all
.  // the data
.
CommOut(); // send some PostScript language data
CommIn(); // check backchannel for PostScript language errors
.
.  // keep calling CommIn() until you have received all
.  // the data
CommEoj(); // send EOJ

/* Terminate the connection if done */
CommClose();
UnLoadComm(); // app supplied routine to unload the comm module
PrClose();

Note This pseudocode does not show any error checking.

Example code that shows how to use the communications module interface is provided in Appendix A. The listing includes the contents of two files: PRGeneral_Comm.h and PRGeneral_Comm.c used to exercise the interface. This sample is intended only to provide insight into the use of the PrGeneral API that allows applications to get the communications resources.
APPENDIX A

A.1 PRGeneral_Comm.c File Contents

/*
   This test application sends down miscellaneous pieces of PostScript language
code. It displays a status window that shows the current status of the
printer. The printjob is cancelled if the user types cmd-period. All data
coming back from the printer are written to a log file.
*/

#include <MacHeaders>
#include <PrintTraps.h>
#include <string.h>
#include <stdio.h>
#include "PrGeneral_Comm.h"

#define kGiveUpTimeInterval30
#define kStatusInterval2*60
#define kLanguageLevelUnknown (-1)

typedef struct {
   long lastWaitNextEvent;
   long lastStatus;
} IdleData;

static pascal OSErr appCommIdle(long idleRefCon, Boolean deadTime);
static void UpdateStatus(void);
static void UpdateError(Str255 m);
static OSErr InLine(CommRecord *comm, Str255 buf);
static Boolean TestCmdChar(void);

/* The macro OutString() is used for outputing C string constants. */
#define OutString(comm, s) CommOut(comm, s, sizeof(s)-1)

/* Test if pat is at the beginning of in. */
#define anchorsearch(in, len, pat) (len >= sizeof(pat)-1 && !memcmp(in, pat,\sizeofof(pat)-1))


TLoadCommProcs ldCommProcs;
TUnloadCommProcs unldCommProcs;
CommRecord *commPtr;

IdleData idleData = {0, 0};
DialogPtr dia;
OSErr gErr = noErr;
Boolean gEoj;

void main()
{
    OSErr err = noErr;
    Str255 buf;
    Boolean eoj;
    FILE *out;
    int languagelevel = kLanguageLevelUnknown; /* assign lang. level as unknown */
    CommOpenInfo openInfo;
    bgInfoRecord bgInfo;
    long i;
    Handle h; /* handle declared for work-around */

    /* Initialization */
    InitGraf(&thePort);
    InitFonts();
    FlushEvents(everyEvent, 0);
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(0L);
    InitCursor();

    MaxApplZone();

    PrOpen();
    /*
    There is no explicit checking of the driver version, since PrGeneral returns
    an error code if the selectors are not defined. A real application should
    have some fallback strategy in this case. For example, send the job with
    PostScriptHandles, using basically the same code as for the background case
    below.
    */
    commPtr = (CommRecord *)NewPtr(sizeof(CommRecord)); /* commPtr is at
    least as large as a CommRecord structure */
    if (commPtr){
        /* load the communications module. */
        ldCommProcs.iOpCode = kLoadCommProcsOp; /* used to get comm resources */
        ldCommProcs.version = kCommProcsVersion; /* tell what version expected */
        ldCommProcs.version = kCommProcsVersion; /* tell what version expected */
ldCommProcs.whichComm = kCommDefault;/* request for default comm channel*/
ldCommProcs.commRecP = commPtr;
ldCommProcs.ignoreBGflag = false;/* driver must check if background print */
PrGeneral((Ptr)&ldCommProcs);/* let the driver load the comm module */

/*
Bug work-around for PSPrinter versions 8.1 and 8.1.1. In these versions of the driver, there is an unlocked resource handle which causes an occasional crash of the system. The crash sometime occurs when PSPrinter's PrGeneral interface to get the communications resources is called and the system is operating with background printing turned on. This problem is not exhibited when operating in foreground. The work-around provided here, gets the PDEF 7 resource handle and locks it after the PrGeneral call is made. This work-around is not needed in version 8.1.2 and subsequent versions of PSPrinter.
*/

if (PrDrvrVers() == 81)
{
    SetResLoad(false);
    h=GetResource('PDEF',7);
    SetResLoad(true);
    if(!h)
        Debugger();
    if (!*h)
        DebugStr("\p Too Bad, handle is already gone.");
    HLock(h);
}

if (ldCommProcs.iError == noErr)
{
    /* at this point the driver has set the appropriate flags in channel field and filled the CommRecord fields. */
    /*
    Now we can call the communications module
    */
    commPtr->idleProc = appCommIdle;/* give driver the app's idle proc */
    commPtr->idleRefCon = (long)&idleData;/* give driver the ptr. to data that may be needed by the app's idle proc */
    out = fopen("BackChannel.log", "w");/* prepare to record printer data */
    dia = GetNewDialog(128, 0, (WindowPtr)-1);/* identify & display status */
    DrawDialog(dia);/*dialog window */
    buf[0] = 0;  /* initialize */
    buf[255] = false;/* flag indicating if we have a full line */
    if (ldCommProcs.whichComm == kCommFile) {/* for other test case */
        /* if saving to disk, call CommOpen with a pointer to a CommLoadInfo
        */
structure that specifies the location, creator and type of the file to be created.

*/
memcpy(openInfo.file.name, "\pDiskFile", 9);
openInfo.file.parID = 0;
openInfo.file.vRefNum = 0;/* Just use lazy search-path */
err = CommOpen(commPtr, &openInfo);
}
else {
    if (ldCommProcs.background) {
        /*
        if not saving to disk and background is on then call CommOpen with a pointer to a bInfoRecord structure (provides more info for PrintMonitor). CommOpen will create a spoolfile.
        */
        PtrToHand("\pPrintFile", (Handle *) &bgInfo.hDocName, 10);
        bgInfo.manualFeed = false;
        err = CommOpen(commPtr, &bgInfo);/* provide more info in background */
    } else {
        /*
        if not saving to disk and background is not on then no spoolfile will be created on call to CommOpen.
        */
        err = CommOpen(commPtr, nil);
    }
}
UpdateStatus();/* update the status dialog window */

/*
    Send some PostScript language code to the printer:
    Note that this code is not DSC conforming.
*/
if (!err) err = OutString(commPtr, "%!r");
if (!err && commPtr->channel.backChannel) {
    /*
    If printing in foreground, we have to the back channel and can directly check the language level and product name of the device.
    */
    err = OutString(commPtr,
"/languagelevel where{pop languagelevel}{1}ifelse ==\r");
    /* if language level is defined, then use the value; if not then device is level 1 */
    if (!err) err = OutString(commPtr, "version == flush\r");
    if (!err) err = CommOut(commPtr, NULL, kFlushWriteBuffers);
    /* remember to flush the buffer */
    UpdateStatus();/* update the status dialog window */

while (!err) {/* notice: that response from the printer might come in small pieces, so we need the while loop for that. */
    /*
We are expecting something from the printer; so keep checking until data is there. Application should have more robust error checking than this.

```
err = InLine(commPtr, buf); /* read some data from the printer */
if (buf[255])
    break;
}
if (!err) {
    --buf[0]; /* remove trailing new line */
    sscanf((char *)buf+1, "%d", &languagelevel);
    fprintf(out, "languagelevel: %#s\r", buf); /* put data into log file */
while (!err) {
    /*
    We are expecting something from the printer; so keep checking until data is there. Application should have more robust error checking.
    */
    err = InLine(commPtr, buf); /* read some data from the printer */
    if (buf[255])
        break;
}
if (!err) {
    --buf[0]; /* remove trailing new line */
    fprintf(out, "version: %#s\r", buf); /* put data into log file */
}
}

if (!err) err = OutString(commPtr, "% begin real job\r\n");
/*
if language level is unassigned in all tests below, then we must be in background or saving to file, so we should send the language level query code with the PostScript stream for both levels.
*/
if (languagelevel == kLanguageLevelUnknown)
if (!err)
    err = OutString(commPtr,
        "/languagelevel where{pop languagelevel 1 eq}{true}ifelse{\r\r};
if (languagelevel == 1 || languagelevel == kLanguageLevelUnknown)
/*
Send down level 1 code.
*/
if (!err) {
    if (!err) err = OutString(commPtr, "% --level1 code--\r\r");
    if (!err) err = OutString(commPtr,
        "/Rectangle { % x y w h Rectangle - rectangle\r\r};
    if (!err) err = OutString(commPtr,
        "\t4 -2 roll moveto % bottom left corner\r\r");
    if (!err) err = OutString(commPtr,
        "\tdup 0.0 exch rlineto % to upper left\r\r");

```
if (!err) err = OutString(commPtr, "\textch 0.0 rlineto % to upper right\r");
if (!err) err = OutString(commPtr, "\tneg 0.0 exch rlineto % to lower right\r");
if (!err) err = OutString(commPtr, "\tclospath \r bind def\r\r");
if (!err) err = OutString(commPtr, "/Rs { % x y w h -or- numarray Rs - rectstroke rect(s) \r"");
if (!err) err = OutString(commPtr, "\tgsave newpath dup type /arraytype eq{aload length 4 idiv \r");
if (!err) err = OutString(commPtr, "\t { Rectangle } repeat \{Rectangle\} ifelse \r"");
if (!err) err = OutString(commPtr, "\t stroke grestore \r }bind def \r\r");
}

if (languagelevel == kLanguageLevelUnknown)
if (!err)
    err = OutString(commPtr, "\r\r");
if (languagelevel == 2 || languagelevel == kLanguageLevelUnknown)
/*
    Send down level 2 code.
*/
if (!err) {
    err = OutString(commPtr, "% --level2 code--\r");
    if (!err) err = OutString(commPtr, "/Rs /rectstroke load def\r\r");
}
if (languagelevel == kLanguageLevelUnknown)
if (!err) err = OutString(commPtr, "\r\r");
if (!err) err = OutString(commPtr, "200 200 100 100 Rs\r");

/*
Output showpage, trailer, send eoj and close connection.
*/
if (!err) err = OutString(commPtr, "showpage\r");
if (!err) err = OutString(commPtr, "%%Trailer\r");
CommEoj(commPtr); /* indicate EOJ to printer, regardless of err */
if (commPtr->channel.eojHandshake) {
    while (!err && !gEoj) 
        err = InLine(commPtr, buf); /* keep reading until we get an EOJ from
                              the printer */
}
CommClose(commPtr); /* terminate the connection, regardless of err */
fclose(out); /* close the log file */

/*
unload the communications module.
*/
unldCommProcs.iOpCode = kUnloadCommProcsOp;
unldCommProcs.version = kCommProcsVersion;
unldCommProcs.commRecP = commPtr;
PrGeneral((Ptr)&unldCommProcs);
DisposePtr((Ptr) commPtr);
}
}
PrClose();
} /* end of main */

//--
This routine represents the applications idle procedure. This procedure gets
called by the driver frequently.
//--
pascal OSErr appCommIdle(long idleRefCon, Boolean deadTime)
{
    IdleData *idleDataP;
    EventRecord theEvent;
    OSErr err = noErr;
    Handle item;
    OSType type;
    Rect box;
    if (gErr)
        return gErr; /* we've already seen some error, so don't update status */
    idleDataP = (IdleData *)idleRefCon;
    if(idleDataP){
        if (deadTime ||
            (idleDataP->lastWaitNextEvent + kGiveUpTimeInterval < TickCount())) {
            WaitNextEvent(everyEvent, &theEvent, 0L, NULL);
            idleDataP->lastWaitNextEvent = TickCount();
            if (TestCmdChar()) {
                err = iPrAbort;
                if (!gErr) {
                    GetDItem(dia, 1, (short *) &type, &item, &box);
                    SetIText(item, "\pCancelling...");
                    gErr = err;
                }
            }
        }
        if (!err && idleDataP->lastStatus + kStatusInterval < TickCount()) {
            idleDataP->lastStatus = TickCount();
            err = CommStatus(commPtr);
            if (!err) {
                UpdateStatus();/* update the status dialog window */
            }
        }
    }
    return err;
} /* end of appCommIdle */
/* This routine tries to read data from printer. buf is a buffer, whose last byte indicates if we received a whole line. This function works even if printer data is broken into pieces, and we can't read a full line in one call to InLine(). */

OSErr InLine(CommRecord *comm, Str255 buf)
{
    OSErr err = noErr;
    static char inBuff[1024];
    static long inBuffLen = 0, inBuffPos = 0;

    if (buf[255]) {
        buf[0] = 0;
        buf[255] = false;
    }
    while (1) {
        if (inBuffPos >= inBuffLen) {
            if (gEoj) {
                buf[255] = true;
                break;
            }
            inBuffLen = sizeof(inBuff);
            err = CommIn(commPtr, inBuff, &inBuffLen, &gEoj);
            if (inBuffLen == 0) {
                break;
            }
            inBuffPos = 0;
        }
        if ((buf[++buf[0]] = inBuff[inBuffPos++]) < ' ' || buf[0] == 254) {
            buf[255] = true;
            break;
        }
    }
    return err;
} /* end of InLine */

/* This routine simply updates the status dialog window with the status received from the printer. */

void UpdateStatus()
{
    Handle item;
    short type;
    Rect box;
    Str255 b;

    GetDItem(dia, 1, &type, &item, &box);
    SetIText(item, commPtr->status);
A.2 PRGeneral_Comm.h File Contents

/*
   This routine simply updates the status dialog window with the error string
   passed as the parameter.
*/
void UpdateError(Str255 m)
{
    Handle item;
    short type;
    Rect box;

    GetDItem(dia, 3, &type, &item, &box);
    SetIText(item, m);
} /* end of UpdateError */

/*
   This routine checks for an end-user request to stop processing with a
   command-period. Notice that this routine is not 'international aware.'
*/
Boolean TestCmdChar()
{
    EventRecord event;

    return GetNextEvent(keyDownMask, &event) &&
        event.what == keyDown &&
        (event.modifiers & cmdKey) &&
        (event.message & charCodeMask) == '.';
} /* end of TestCmdChar */

A.2 PRGeneral_Comm.h File Contents

#ifdef __PrGeneral_Comm__
#define __PrGeneral_Comm__

enum {
    kLoadCommProcsOp= 16,
    kUnloadCommProcsOp= 17
};

#define kCommProcsVersion 1
#define kFlushWriteBuffers(-1)
The following error code is returned in the ‘ioError’ field of the TLoadCommProcs structure.

```c
enum {
    kUnsupportedVersion = 10
};
```

When calling the kLoadCommProcs PrGeneral routine, fill in the ‘whichComm’ field of the TLoadCommProcs structure with one of the following values:

```c
typedef enum {
    kCommDefault = 0, // Get the driver’s default channel
    kCommFile      // Get the driver’s save to disk channel.
} CommSelector;
```

```c
typedef struct TLoadCommProcs {
    short iOpCode; // input, must be kLoadCommProcs.
    short iError; // output.
    long lReserved; // reserved for future use.
    short version; // input/output.
    unsigned short ignoreBGflag: 1; // input.
    unsigned short background : 1; // output.
    unsigned short reserved : 14;
    CommSelector whichComm; // input.
    struct CommRecord *commRecP; // input; structure gets filled in by the driver.
} TLoadCommProcs;
```

```c
typedef struct TUnloadCommProcs {
    short iOpCode; // input, must be kUnloadCommProcs.
    short iError; // output.
    long lReserved; // reserved for future use.
    short version; // input/output.
    struct CommRecord *commRecP; // input.
} TUnloadCommProcs;
```

```c
typedef struct bgInfoRecord {
    StringHandle hDocName;
    unsigned short manualFeed : 1;
    unsigned short reserved : 15;
} bgInfoRecord;
```

When opening a kCommFile channel, pass a pointer to this structure to
CommOpen(). This structure specifies the location, creator, and type of the file to be created. If a kCommDefault channel is being opened then pass NULL for the CommOpen() 'openInfo' parameter.

typedef struct CommOpenInfo {
    FSSpec file; // Where should the file be located.
   OSType creator; // Creator for the new file.
    OSType type; // Type of the new file
} CommOpenInfo;

typedef struct ChannelCaps {
    short reserved : 12; // These aren't used yet.
    short backChannel : 1; // Can we get info from the printer
    short transparent : 1; // Handles control codes without quoting.
    short eightBit : 1; // The channel supports 8 bit characters.
    short eojHandshake : 1; // True if client must wait for printer's eoj.
    long responseTimeOut; // Maximum time it will take printer to responed (Ticks)
} ChannelCaps;

typedef pascal OSErr (*CommOpenProcPtr)(struct CommRecord* comm, void *more);
typedef pascal OSErr (*CommInProcPtr)(struct CommRecord* comm, void *buffer,
    long *numBytes, Boolean *eoj);
typedef pascal OSErr (*CommOutProcPtr)(struct CommRecord* comm, void *buffer,
    long numBytes);
typedef pascal OSErr (*CommEojProcPtr)(struct CommRecord* comm);
typedef pascal OSErr (*CommStatusProcPtr)(struct CommRecord* comm);
typedef pascal OSErr (*CommCloseProcPtr)(struct CommRecord* comm);

typedef pascal OSErr (*CommAppIdleProc)(long idleRefCon, Boolean deadTime);

typedef struct CommRecord {
    CommOpenProcPtr commOpen;
    CommInProcPtr commIn;
    CommOutProcPtr commOut;
    CommEojProcPtr commEoj;
} CommRecord;
CommStatusProcPtr commStatus;
CommCloseProcPtr commClose;

ChannelCaps channel; // Information about the channel.
Str255 status; // The current status of the channel.
Handle dataH; // The comm code hangs its data structure here.

CommAppIdleProc idleProc;
long idleRefCon; // Passed to the caller’s idle proc.

char reserved[32]; // Reserved by the driver (private fields)
} CommRecord;

/*@*/

enum {
  kBadCommPtrErr = -4000
};

/*@*/

#define CommOpen(comm, openInfo)
  (((comm) && (comm)->commOpen) ? (*(comm)->commOpen)((comm), (openInfo))
   : kBadCommPtrErr)
#define CommIn(comm, buffer, nBytes, eoj)
  (((comm) && (comm)->commIn) ? (*(comm)->commIn)((comm), (buffer), (nBytes),
    (eoj)) : kBadCommPtrErr)
#define CommOut(comm, buffer, nBytes)
  (((comm) && (comm)->commOut) ? (*(comm)->commOut)((comm), (buffer), (nBytes))
    : kBadCommPtrErr)
#define CommEoj(comm)
  (((comm) && (comm)->commEoj) ? (*(comm)->commEoj)((comm))
    : kBadCommPtrErr)
#define CommStatus(comm)
  (((comm) && (comm)->commStatus) ? (*(comm)->commStatus)((comm))
    : kBadCommPtrErr)
#define CommClose(comm)
  (((comm) && (comm)->commClose) ? (*(comm)->commClose)((comm))
    : kBadCommPtrErr)
#endif
Index

A
AppleTalk 6

B
backChannel 7
background 7, 9, 10
bgInfoRecord 9
buffer 10, 11

C
ChannelCaps 10
CommClose 7
CommClose() 12
commEoj 7
CommEoj() 12
commIn 7
CommIn() 11
commOpen 7
CommOpen() 9
commOut 7, 10
CommOut() 10
CommSelector constants 6
commStatus 7
CommStatus() 12

D
deadTime 8

E
eightBit 10
end of job 11, 12
EOJ 11, 12
eoj 11
eojHandshake 11
EPS 7

F
EventEvail() 8
flush 11

G
GetNextEvent() 8
Getting Status from the Printer 12

I
idleProc 8
idleRefCon 8
ignoreBGflag 7
Indicating the End of Job to the Printer 12
Initiating Communication 9

K
kCommDefault 6, 9
kCommFile 6, 9
kCommProcVersion 6
kFlushWriteBuffers 10
kLoadCommProcsOp 5, 6, 7
kUnloadCommProcsOp 5, 8

L
LaserJet 5
LaserWriter 5
Loading the Communications module 6

N
numBytes 10, 11
PICT  7
PrClose()  12
PrDrvrvs()  5
PRGeneral_Comm.c  13
PRGeneral_Comm.h  7, 13
PrintMonitor  7
Pseudocode of a Typical Job  12
PSpict2eps  7
PSPrinter  5

R
Reading Data From the Printer  11

S
Sending Data Down the Channel  10
Status  12

T
Terminating the Connection  12
TLoadCommProcs  6, 7
transparent  10

U
Unloading the Communications
module  8
userCanceled()  9
Using kLoadCommProcsOp and
kUnloadCommProcsOp  6

W
WaitNextEvent()  8
whichComm  9