The PPR Hacker's Guide

The PPR Hacker's Guide

David Chappell

7 November 2003

Table of Contents

Abstract

This document provides information of interest to those wishing to modify the PPR source code or to add plugin modules such as printer interface programs and programs that send messages to users.

Configuration File Formats

Configuration files which the system administrator is expected to modify directly are described within the reference manual. The files described in this section are modified automatically by programs such as **ppad**. That is why they are described here in the Hacker's Guide.

Format of a Printer Configuration File

Each file in the directory `/etc/ppr/printers/` represents one printer. It is permissible to create and edit these files manually, however the command **ppad** has been provided to do this automatically.

A new printer configuration file is created by the **ppad interface** command. The new configuration file is created by turning the arguments of the **ppad interface** command into a `Interface:` and an `Address:` line and appending the file `/etc/ppr/newprn.conf`. The `newprn.conf` file is normally created by the **ppad new alerts** command.

Any lines begining with `;` or `#` are comments. Blank lines are ignored. All other lines should begin with a keyword which is followed by one or more values. Only the `Bin:` and `PPDOpt:` lines should occur more than once. If any other line occurs multiple times, all but the last instance are ignored.

Any line that begins with an unrecognized keyword is ignored. This is so that you can add lines of your own which will be understood and acted on by other programs. The program **ppr2samba** works on this principle.

`Comment: string`

This line contains a comment which describes the printer. The comment is displayed when the **ppad show** command is used to display the printer configuration. This line is optional.

`Interface: interface`

This line is required. It gives the name of the program which the spooler should invoke in order to make contact with the printer. If the *`interface`* does not begin with a slash, then it refers to a subdirectory of the `/usr/lib/ppr/interfaces/` directory. Common values for this line are `atalk`, `tcpip`, and `serial`.

`Address: address_string`

The address string is passed to the interface program as its second parameter. The proper format for this string depends on the interface. Reasonable values for use with the interfaces named in the above paragraph are `Mac Laser Printer: LaserWriter@MYZONE`, `smith.prn.myorg.org:9100`, and `/dev/ttys05`. If the address contains leading or trailing spaces it should be enclosed in double quotes. Technically, this line is not required, but the default address is the name of the printer which is unlikely to be useful.

`Options: string`

This line is optional. The value *`string`* is passed to the interface as its third parameter.

`PPDFile: filename`

This line specifies the path and name of an Adobe Post Script Printer Description file which describes the printer. If the name does not begin with a slash, it referes to a file in the directory `/usr/lib/ppr/PPDFiles`. This line may not be required but it really ought to be present.

`Alert: interval method address`

This line is optional. If it is present, then messages will be sent to the designated person when faults occur on the printer.

The integer *`interval`* indicates how often these messages will be sent. For instance, if the value is `5`, then a message will be sent for every 5th fault.

The *`method`* parameter indicates the method by which the message should be sent. Currently, this parameter is ignored. Set it to `mail`.

The *`address`* indicates the person to whom the message should be sent. Since `mail` is currently the only supported method, this should be a e−mail address.

`FlagPages: integer integer`

This line is optional. The first integer refers to banner pages, the second to trailer pages. Four different values are allowed. They are `0` for 'never', `1` for 'preferably not', `2` for 'preferably yes', and `3` for 'always'.

`Feedback: boolean`

This line is optional. It indicates whether or not the connexion to the printer allows it to send messages back to the spooler. There is a list of default values for all the interfaces supplied with PPR compiled into **pprdrv**. (This list is defined in `include/interfaces.h`.) If this line is not present and the interface is not in the list, it is assumed to be `false`. Any instances of this line which occur before the `Interface:` line are ignored.

`JobBreak: integer`

This line is optional. The integer is the code number of a job break method. This line is only necessary if a job break method other than the default for the interface is desired. Default job break numbers for the interfaces which come with PPR are defined in the source file `include/interfaces.h`. The meaning of the values in `interfaces.h` can be understood by reading the **ppad jobbreak** section of the ppad(8) man page. Any instances of this line which occur before the `Interface:` line are ignored.

`Codes: integer`

This line is optional. It indicates the range of character codes which the interface can transmit to the printer. The acceptable values are `1` for Clean7Bit, `2` for Clean8Bit, `3` for Binary and `4` for Binary if TBCP is used.

`GrayOK: boolean`

This line is optional. The default value is `true`. If this parameter is set to `false`, the printer will refuse to print any job which does not have `color` in a `%%Requirements:` line in its header. A `GrayOK:` line can be used to prevent black−and−white or grayscale jobs from begin printed on an expensive colour printer.

`Charge: money [money]`

This line is optional. If this line is present, then the printer becomes a protected printer. Each parameter *`money`* will normally be a positive number with two decimal places. If both are `0.00`, then the printer is protected, but no actual charges are mode to the user's account. The first number is the amount that should be charged for each sheet of paper printed on both sides. The second is the amount that should be charged for each sheet printed on only one side. If the second number is missing it is assumed to be the same as the first.

`OutputOrder: direction`

This line is optional. If it is absent, it is assumed that direction is `Normal`. If direction is set to `Reverse`, the spooler will cause the pages to be printed in reverse order, if possible. The value of this line can be set with the command **ppad outputorder printer Normal**, **ppad outputorder printer Reverse**, or **ppad outputorder printer ppd**. The command **ppad outputorder printer ppd** deletes any `OutputOrder:` line which may exist.

`Bin: binname`

The *`binname`* parameter is the bin name as it appears in a `*InputSlot` line of the PPD file. Common values are `Upper`, `Lower`, and `Cassette`. If automatic bin selection and media handling are desired, there should be one `Bin:` line for each installed bin. Removing all the `Bin:` lines disables automatic bin selection and media handling.

`DefFiltOpts: options`

This line is optional. If present, it contains input filter options in the form of name−value pairs. (If the user employs one or more `−o` switches when submitting a job with **ppr**, then the argument of each `−o` switch is appended to this list. Thus, `−o` switches can be used to override settings in this list.) This line is automatically updated whenever the **ppad ppd** command is used to select a new PPD file or the **ppad ppdopts** command is used to change the PPD options. An update may be forced at any time with the **ppad deffiltopts** command.

`Switchset: switch_description`

This line is optional. If present, it contains a compressed description of the switch settings which were saved with the **ppad switchset** command.

`PassThru: typelist`

This line is optional. If present, it contains a space separated list of file types which should be passed directly through to the printer.

`PagesLimit: integer`

This line is optional. If present it indicates the maximum number of pages this printer is allowed to print in a single job. Jobs with more pages than the number indicated are rejected. If the job was submitted to a group, it may be printed on another member of the group. If no printer will print it, then it is arrested.

This line may be edited with the **ppad pageslimit** command.

`PageCountQuery: integer`

This line can be used to enable pair of queries which fetch the printers lifetime page count before and after the printing of the job (exclusive of banner and trailer pages).

The information obtained through these queries can be logged. See Section ???.

If this line is absent or the *`integer`* is 0, then this feature is disabled.

If the value is 1, then then the PostScript code **statusdict /pagecount get exec ==** is used to get the page count. No attempt is made to make sure that the print engine has come to a stop and the printer has updated the count before making the query. On some printers this does not cause a problem because the page count is updated imediately. On others it may be solved by using the `pjl` or `signal/pjl` jobbreak method which causes PPR to wait until all of the pages have hit the output bin before considering the job complete.

In the future, additional values for this parameter may be defined. These additional values will use different techniques to obtain the page count.

`Commentator: number name address options`

A printer configuration file may have zero or more of these lines. Their purpose is explained in Appendix ???.

`PPDOpt:` `option` `value` `(description)`

A printer configuration file may have zero or more of these lines. Each line describes the setting of the options listed in the PPD file. These settings generally describe optional equipment which may be installed in or attatched to the printer such as additional paper trays or duplex attachements. The *`option`* is the option name from the PPD file, without the translation string (the part which may follow a slash). The *`value`* is one of the possible option values listed in the PPD file, again without the translation string. The description is formed by combining the translation strings for the option and the value. (Option settings and the related terminology are explained in Adobe's PostScript Printer Description File Format Specification.) These lines can be generated automatically with the **ppad ppdopts** command.

`ppr2samba:` `include` `prototype`

Used by **ppr2samba**. See the man page ppr2samba(8).

`ppr2samba-drivername:` `name`

Used by **ppr2samba**. See the man page ppr2samba(8).

`ms-driver-name:` `name`

Formerly used by **ppr2samba**. See the man page ppr2samba(8).

Format of a Group Configuration File

Each file in the `/etc/ppr/groups/` directory represents a group. It is permissible to create and edit these files manually, however it is generally more convient to use the command **ppad** to do it automatically. The `Printer:` line is the only one which should appear more than once. If any other line occurs more than once, all but the last instance are ignored. Lines with `;` or `#` in the first column are comments. The remaining lines are of two types:

`Comment:` `string`

This line contains a comment describing the group. This comment is displayed by the **ppad group show** command. This line is optional.

`Printer:` `name`

The *`name`* is the name a printer that should be a member of the group. There should be one line for each group member.

`Rotate:` `boolean`

This value indicates if the spooler should attempt to distribute the load evenly among the printers. If it is `False`, the spooler will always take the first idle printer in the group. If it is `True`, the spooler will attempt to use each printer in turn. The default is `True`.

`DefFiltOpts:` `options`

This line is optional. If present, it contains some name−value pairs to be prepended to any the user supplies with **ppr**'s **−o** switch. This line will be automatically updated by **ppad** whenever a group member is added or deleted.

`Switchset:` `switch_description`

If this line is present, it contains a compress representation if the switches saved with the **ppad group switchset** command.

`PassThru:` *`type_list`*

This line is optional. If present, it contains a space separated list of file types which should be passed directly through to the printer.

`ppr2samba:` *`include prototype`*

Used by ppr2samba. See the man page ppr2samba(8).

PPR's Directories and Files

This section describes the directory structure found within PPR's home directory (`/usr/lib/ppr/`), in its spooling directory, (`/var/spool/ppr/`) and in its configuration directory (`/etc/ppr/`).

The `/var/spool/ppr/` Directory

The `/var/spool/ppr/queue/` Directory

The `/var/spool/ppr/queue/` directory contains one file for each print job currently in the queue. Each file name is the same as the queue id of the job is describes. These files are created by the program **ppr** and removed by the print daemon **pprd** when the job has been printed or is canceled.

The `/var/spool/ppr/jobs/` Directory

The `/var/spool/ppr/jobs/` directory contains three to five files for each job currently in the queue.

The first part of each file name is the queue id of the job. The file whose name ends in `-comments` contains any header and trailer comment lines which have not been removed and represented by parameters in the file in the queue directory.

The file whose name ends in `-pages` contains the text of the document default section, if it exists, and a list of the pages in the document. The record for each page includes the offset in the `-text` file at which it begins and any page header and trailer comments for that page.

The file whose name ends in `-text` contains most of the text of the job. If the `-S true` switch was used when **ppr** was invoked, then resources will be missing from this file, having been stript out and replaced by comments which will later be used to put them back.

The files with names ending in `-log` contain the print job logs. If **ppr** was invoked with the `-w log` switch then any warnings will be in this file. Any text received from the printer, including printer error messages, will also be in this file. Messages which explain why the job was routed away from a particular printer will be in this file, but each line of such messages will begin with a "+". Whenever a banner or trailer page is printed, the contents of the log file is printed and the log file is deleted.

The `/var/spool/ppr/printers/alerts/` Directory

Each time a fault message is generated for a printer, it is appended to a file in this directory. There is one file for each printer. If the file is more than one hour old, the file is truncated to zero length before writing the new alert instead of appending. This ensures that only recent messages will be sent to the operator.

The `/var/spool/ppr/printers/status/` Directory

This file may contain one file for each printer. If present, the file contains the last status message received from the printer.

The `/var/spool/ppr/logs/` Directory

This directory contains log files. Error messages may be written in these log files when PPR component fail. If this directory contains the file `printlog`, PPR will append a line for each file printed. (See Appendix ???.)

The `/etc/ppr/` Directory

This directory contains a number of configuration files. In also contains subdirectories which will be described in later sections.

The file `papsrv.conf` is descibed in the papsrv.conf(5) man page.

The purpose of `newprn.conf` is explained in Appendix Section , Format of a Printer Configuration File .

The file `smb−include.conf` is a segment of Samba configuration file. It is generated by the program **ppr2samba**.

The file `charge_users.db` is the user charge accounts database. The program **ppuser** can be used to read and modify this file.

The documentation for the file `lw−messages.conf` can be found in `lw−messages.conf.sample`.

The file `media.db` is the list of known media types. It is consulted by **ppr** and **pprdrv**. It may be read and modified by the **ppad media** series of commands.

The `/etc/ppr/` Directory

The `/etc/ppr/printers/` Directory

There is one file in this directory for each printer. Each file has the same name as the printer it describes. The format of one of these files is described in Appendix Section , Format of a Printer Configuration File .

The `/etc/ppr/groups/` Directory

There is one file in this directory for each group. Each file has the same name as the group it describes. The format of these files is described in Appendix Section , Format of a Group Configuration File .

The `/etc/ppr/mounted/` Directory

This directory contains one file for each printer. Each file contains a list of a printer's bins and the medium mounted on each bin. This file is created by the main print daemon **pprd**. When **pprd** starts up it looks in this directory to determine what media was mounted last time it was running. It automatically re−mounts all those media.

The program **pprdrv** also uses this file in order to select bins for printing banner pages and to automatically select bins for print jobs according to their required media.

The `/usr/lib/ppr/` Directory

The `/usr/lib/ppr/bin/` Directory

This directory contains the program **ppr** which submits jobs, various utility programs such as **ppop** and **ppad**, the PPR daemons **pprd**, and the **papsrv**.

The `/usr/lib/ppr/lib/` Directory

This directory contains **pprdrv** and as well as other programs a user or administrator wouldn't normally execute directly. It also contains some configuration files which even a system administrator wouldn't ordinarily modify. These files would be modified by people who are creating new components for PPR.

The documentation for the file file `lw-messages.conf` is in the file itself.

The file `mfmodes.conf` is used by **ppad** when determining the default filter options for a printer or group. The format of this file is described in Section ???.

The file `fontsub.conf` is used to find substitutes for missing fonts. It is described in Section , Font Substitution Configuration File .

The `/usr/lib/ppr/interfaces/` Directory

Each file in this directory is a program which is responsible for making contact with any printer which has a particular kind of interface. For instance, `interfaces/atalk` is used to communicate with printers connected through Apple's Printer Access Protocol. If you would like to write a printer interface program, see section Section , Printer Interface Programs .

The `/usr/lib/ppr/responders/` Directory

Each file in the `responders/` directory is a program which can be used to attempt to send a message to the user who submitted the job. A responder program is selected at the time the user invokes **ppr**, by using the `-m` switch. The parameter for the `-m` switch is the name of the program in the `responders/` directory which should be used. If the `-m` switch is not used, the default responder, `write`, is used. (You can change the default by setting the environment variable `PPR_RESPONDER`.) The manner in which a responder is invoked is described in Appendix Section , Requirements for a Responder .

The `/usr/lib/ppr/commentators/` Directory

This directory contains small programs similar to responders. Commentators will soon be merged with responders, so don't worry about this directory.

The `/usr/lib/ppr/filters/` Directory

The programs in this directory are filters which convert various file formats to PostScript. If you would like to write a new filter, see section Section , Input Filters .

The `/usr/lib/ppr/fixup/` Directory

This directory contains skeletal scripts which could potentially be installed in `/usr/lib/ppr/filters/` by the command **ppd−index filters**.

The `/usr/lib/ppr/cgi−bin/` Directory

Files in this directory are part of the PPR web interface. This directory appears as `/cgi−bin/` at the top level of the PPR web server's directory tree.

The `/usr/share/ppr/` Directory

The `/usr/share/ppr/cache/` Directory

This directory contains the PostScript resources distributed with PPR. They are segregated in subdirectories by resource type. Many of the input filters distributed with PPR use resources in the `encodings/` and `procset/` subdirectories. The filters don't use these files directly. Instead, they insert `%%IncludeResource:` in their output so that **pprdrv** will insert the contents of the desired file.

The `/usr/share/ppr/www/` Directory

This directory is the root for the PPR web server **ppr−httpd**.

The `/usr/share/ppr/fonts/` Directory

This directory contains fonts and font metric files which are distributed with PPR. Formerly, the font files were stored in `/usr/share/ppr/cache/font/`, but the font index mechanism made this unnecessary.

The `/usr/share/ppr/man/` Directory

This directory contains the PPR man pages. Packagers may choose to relocate this files to the system−wide man page directory. If not, one must either use the **ppdoc** command to view them or add this directories to one's `MANPATH`.

The `/usr/share/ppr/PPDFiles/` Directory

This is the default location for PPD files. It contains a collection of PPD files for common printers. Most of these were provided by the manufacturers or by Adobe[1] but a few are part of PPR.

The `/usr/share/ppr/locale/` Directory

This directory contains GNU Gettext message catalogs. These catalogs are used to display program messages in languages other than English. If you would like to add messages for your language, see the file `README.txt` in the source code directory `po/`.

The `/usr/share/ppr/speach/` Directory

This directory contains sound files used by the audio responder.

Printer Interface Programs

This section provides the information you will need to write a PPR printer interface program. An interface is a program, possibly a shell script, which receives the text of the print job from **pprdrv** and sends it to the printer. The interface program will receive the print job text on stdin.

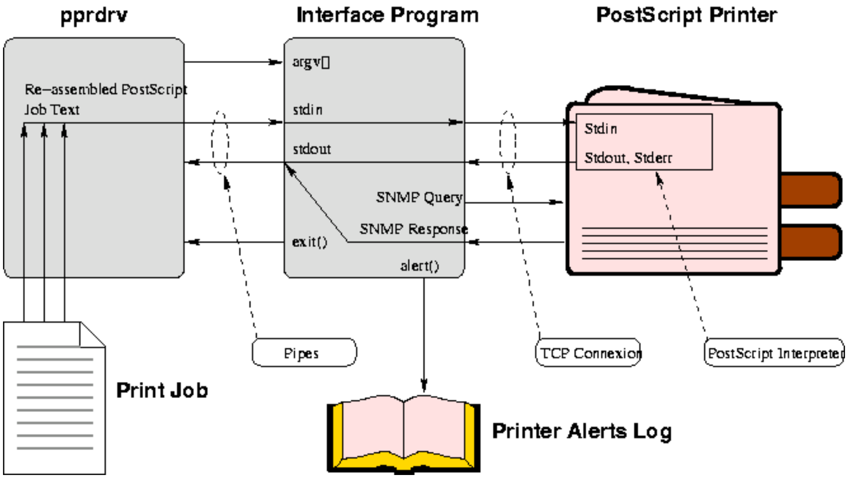
The interface program should should print any data received from the printer on stdout which will cause them to be sent to **pprdrv** for analysis. The interface program may print additional specially formatted messages on stdout in order to inform **pprdrv** of its progress or the current status of the printers. Any messages which **pprdrv** does not recognize as a progress or status message will be appended to the print job's log file.

The interface program will receive command line parameters which include the name of the printer, the printer's address, interface program options, and other parameters which describe certain aspects of **pprdrv**'s intended interaction with the printer.

The interface program may append messages to the printer's log file by calling the `alert()`. It should do this to report invalid command line parameters or unexpected problems communicating with the printer.

The interface program must return an exit codes in order to inform **pprdrv** of its success or failure. If the code indicates failure, then **pprdrv** will adopt the exit code as its own. If the code indicates success, then **pprdrv** will adopt it only if it too achieved success.

Figure 1. A typical Printer Interface Program acts as an intermediary between pprdrv and a PostScript printer. It communicates with the printer by using TCP/IP and SNMP.



Some interface programs have a special probe mode. In probe mode they don't connect to the printer, but instead use out−of−band means to extract identifying information from the printer. An interface program's probe mode is one of the ways the **ppad ppdq** and **ppad ppd query** commands attempt to identify the printer in order to recommend an appropriate PPD file.

In the remainder of this sections, the way an interface program ought to work is explaned in more detail.

The Command Line Parameters

There are 10 parameters in all. That seems a lot, but very few interface programs will examine any beyond the first six. Most will use parameters one through three and simply verify that four through six have acceptable values. It is perfectly acceptable to write an interface which uses only the first two.

It is helpful if the interface program also examines parameters four through six, if for nothing else, to make sure that it and the communications channel which it opens to the printer are able to cope with the implications of the values of these parameters. If it cannot, it should call `alert()` to post an explanatory message to the printer's alert log and then exit with the code EXIT_PRNERR_BAD_

Some of the parameters are small integers which represent enumerated values. These values are defined symbolically in the C include file `include/interface.h` and in the shell script fragment `interface.sh`.

Parameter One−−The Printer's Name

The interface is invoked with the first parameter set to the name of the printer. The interface should use this name when posting alerts using either the alert() function in libppr.a or lib/alert.

Parameter Two−−The Printer's Address

The second parameter is set to the string following the `Address:` keyword in the printer configuration file or to the printer name if there is no address line. (That last feature is a holdover from very old versions of PPR. Printer configuration files created by **ppad** always have `Address:` lines.)

What constitutes a syntatically valid address is entirely up to the designer of the interface program. If the address is syntactically invalid, the interface must call `alert()` to post an error message to that effect and then exit returning the code EXIT_PRNERR_NORETRY_BAD_SETTINGS.

Parameter Three−−The Interface Options

The third parameter is set to the string following the `Options:` keyword in the printer's configuration file. This will be zero or more space separated name=value pairs.

The set of valid options and acceptable values is determined by the designer of the interface program. However, when devising options for a new interface program, he should be aware of the options of similiar existing interfaces and avoid creating new options or option sementics. If the interface program does not recognize one of the options or finds an option with an invalid or out−of−range value, it must call `alert()` to post an error message to that effect and then exit with the value EXIT_PRNERR_NORETRY_BAD_SETTINGS.

Parameter Four−−The JobBreak Value

The fourth parameter is the value from the printer configuration file's `JobBreak:` line. This value is a small integer representing an enumerated value. The `JobBreak:` line is set with the command **ppad feedback**.

Most job break values do not require any special action on the part of the interface. However, it is a good idea for the interface program to test for jobbreak settings with which it is known to be incompatible. It should only check for settings *known* to be incompatible. It should *not* limit jobbreak settings to a list of those known to be compatible since other, compatible settings may be added in future versions of PPR. The various possible jobbreak settings are described in the ppad(8) man page under the section for the **ppad jobbreak** command. Most interfaces will only be incompatible with **JOBBREAK_SIGNAL** and **JOBBREAK_SIGNAL_PJL**.

The jobbreak methods **JOBBREAK_SIGNAL** and **JOBBREAK_SIGNAL_PJL** require explicit support in the interface program. The interface **atalk** is the only one supplied with PPR which does this. If it detects that one of these jobbreak methods is in use, it will send its parent (**pprdrv**) SIGUSR1 as soon as it has established its own SIGUSR1 handler. Thereafter, whenever it receives SIGUSR1, it will read all bytes currently available from the pipe, send them to the printer and then send the printer an end of job indication. When **atalk** receives an acknowleding end of job indication from the printer, it should sends **pprdrv** SIGUSR1. This handshaking method is necessary because the PAP (AppleTalk Printer Access Protocol) end of file marker has no representation in the byte stream. Instead it is sent out−of−band by setting a special flag in the header of the packet.

Parameter Five−−The Feedback Value

The fifth is the value from the printer configuration file's `Feedback:` line. It is zero if the printer is incapable of sending data back to the computer running PPR over the communications line, non−zero if it is capable. The `Feedback:` line is set with the **ppad feedback** command.

Parameter Six−−The Codes value

The sixth parameter is the value from the printer configuration file's `Codes:` line. The value is a small integer representing an enumerated value. The `Codes:` line is set with the command **ppad codes**.

This parameter indicate the set of byte values which **pprdrv** believes that the interface program and the communications channel are capable of conveying all the way to the PostScript interpreter. It is recommended that the interface program examine the codes value and abort if the interface and the communications channel are not capable of passing the set of codes indicated.

Parameter Seven−−The Job Name

The seventh parameter is set to the name of the job. (For example, `mouse:chipmunk−148.0(mouse)`. This parameter will be of no interest to most interface programs. An exception is the **lpr** interface which passes this information on to the remote system.

Parameter Eight−−The Routing Instructions

The eight parameter is set the the text of the `%%Routing:` comment in the print job. If the `%%Routing:` comment is absent, then this parameter will be blank. The routing comment is intended to convey written instructions to a human printer operator. For example, its value might be "deliver to room 101". If your interface program can somehow make this message appear on an operator console, it may do so. If it is an interface to a fax server, then it may read the telephone number from this parameter. This is acceptable because a telephone number can be considered a delivery instruction. This parameter should not use this parameter for any other purpose. Particularly, it should not be used as a way to pass options which control the printing process.

Parameter Nine−−The User's Name

The ninth parameter is the text of the `%%For:` comment in the print job. (The `%%For:` line in the job as sent to the printer which is not necessarily the same as the `%%For:` line in the input file.) If this information is unavailable, the field is blank. The **lpr** interface passes this on to the remote system. Unless your interface passes the job on to another spooler (as the lpr interface may), it should ignore this parameter.

Parameter Ten−−The File Type

The tenth was once used by special interface programs which ran the input file through Ghostscript if this parameter was empty. Since **pprdrv** now performs this function, these interface programs are are obsolete. This parameter will probably be removed in a future version of PPR. This only possible reason for preserving it is if someone can show that some printer comuncation method requires the file type to be communicated out−of−band.

The tenth parameter indicates the type of the file on stdin, that is, the file which the interface program is begin asked transmit to the printer. Normally this parameter is an empty string, which indicates that the file is in PostScript format.

If passthru mode is in effect (see **ppad passthru** and **ppad group passthru**), that is if the file to be transmitted to the printer is not a PostScript file, then this parameter will be the name of the file type, such as `pcl` or `pdf`. (The file type names which appear in this parameter are the same names as are used with the **ppr −T** switch. A non−empty tenth parameter *does not* necessarily indicate that a **−T** switch was used.)

Also, if transparent mode is in effect (see **ppr −H transparent**), then this parameter will contain a space separated list of the filters which would be required (and may actually have been invoked) to convert the input file to PostScript. Uncompression with **uncompress** and **gunzip** are indicated by the strings `uncompress` and `gunzip` respectively. Here are some examples: if the input file is already PostScript, then this parameter will be empty. If the file is PCL data, then this paramater will be `pcl`. If the file is **gzip** compressed PostScript, then this parameter will be `gunzip`. If the file is **gzip** compressed PCL, then this parameter will be `gunzip pcl`.

Help in Parsing the Command Line

The interface command line in not particularly difficult to parse, but you might find it helpful to use the library functions which those interfaces which come with PPR and are written in C use. To use it, include the file `libppr_int.h` and call the function `int_cmdline_set()` from `main()` like this:

```
int_cmdline_set(argc, argv);
```

Once you have do this, the parameters will be available to you as members of the global structure `int_cmdline`. The structure `int_cmdline` is an instance of `INT_CMDLINE`, which is defined below:

```
struct INT_CMDLINE
        {
        gu_boolean probe;               /* TRUE if −−probe used */
        const char *int_name;           /* example: "interfaces/atalk" */
        const char *int_basename;       /* example: "atalk" */
        const char *printer;            /* example: "myprn" */
        const char *address;            /* example: "My Laser Printer:LaserWriter@Computing Cente
        const char *options;            /* example: "idle_status_interval=60 open_retries=5" */
        int jobbreak;                   /* example: 1 (JOBBREAK_SIGNAL) */
        gu_boolean feedback;            /* example: 1 (TRUE) */
        enum CODES codes;               /* example: 3 (CODES_Binary) */
        const char *jobname;            /* example: "mouse:myprn−1001.0(mouse)" */
        const char *routing;            /* example: "Call David Chappell at 2114 when ready" */
        const char *forline;            /* example: "David Chappell" */
        const char *barbarlang;         /* example: "" (PostScript) */
        const char *title;              /* example: "My Print Job" */
        } ;
```

It is now very easy to write code that refers to the command line parameters by name. For example, if your interface program doesn't support probe mode, bidirectional communication, or the jobbreak methods signal and signal/pjl, you could put this right after the call to `int_cmdline_set()`:

```
if(int_cmdline.probe)
        {
        fprintf(stderr,
                _("The interface program \"%s\" does not support probing.\n"),
                int_cmdline.int_basename
                );
        exit(EXIT_PRNERR_NORETRY_BAD_SETTINGS);
        }

if(int_cmdline.feedback)
        {
        alert(int_cmdline.printer, TRUE,
                _("The PPR interface program \"%s\" is incapable of sending feedback."),
                int_cmdline.int_basename
                );
        exit(EXIT_PRNERR_NORETRY_BAD_SETTINGS);
        }

if(int_cmdline.jobbreak == JOBBREAK_SIGNAL || int_cmdline.jobbreak == JOBBREAK_SIGNAL_PJL)
        {
        alert(int_cmdline.printer, TRUE,
                _("The jobbreak methods \"signal\" and \"signal/pjl\" are not compatible with\n"
                "the PPR interface program \"%s\"."), int_cmdline.int_basename);
        exit(EXIT_PRNERR_NORETRY_BAD_SETTINGS);
        }
```

The Return Codes

The exit codes which interface programs should use are defined in `include/interface.h` and `/lib/interface.sh`.

EXIT_PRINTED
: This interface program should return this code if it was able to complete its jobs sucessfully. It doesn't actually mean that the interface knows the job was printed sucessfully, just that it was able to do its part. When it receives this exit code, **pprdrv** will exit with the same code if the interface program accepted all of the job data before exiting and if no other error was detected (such as a PostScript error).

EXIT_PRNERR
: If the interface was unable to connect to the printer or the connection was broken off, it may return this code. The printer will be fault−auto−retry mode. This is the catchall code. If there is a code listed below which better fits the specific circumstance, it would be better to return that code.

EXIT_PRNERR_NORETRY

There was a printer error caused by a circumstance which will not disappear spontainiously, such as a syntactically invalid printer address. The printer will be placed in fault−no−auto−retry mode. Like EXIT_PRNERR, this is a catch−all code. If there is a code listed below that better fits the specific circumstance, then it would be better to return that code.

EXIT_JOBERR

This code is normally only used by **pprdrv**. Very few interfaces would ever have cause to return this error code.

Since PostScript errors are detected by **pprdrv** by watching for messages from the PostScript interpreter in the data stream which the interface receives from the printer and prints on stdout, there is no need for the interface to detect PostScript errors or to report them using this exit code.

But, if the interface detects an error and has some kind of information which definitely indicates that the fault was caused by the job being printed, it should return this code. The job will be "arrested" and held for inspection by the operator.

One of the few types of interfaces which might return this code is an interace to a fax server. It might return this code if the phone number (possibly passed as a routing instruction) turned out to be invalid. In such a case it could print a message to that effect to stdout (which leads into the jobs log file) and then return this code.

EXIT_SIGNAL

When a job is canceled during printing or the printer is forced to halt during printing, **pprdrv** sends SIGTERM to the interface program. The interface program has the option of catching the signal and performing shutdown operations before termination. After the shutdown is finished, it should return this code. Note that an interface program is not required to catch the signal. Simply dieing is perfectly acceptable.

EXIT_ENGAGED

If the interface was unable to connect to the printer because it was busy or off−line, it should return this code. The printer state will be set to "otherwise engaged or off−line" and the operation will be retried after a short delay.

Normally this code will be used when a definite indication is present that the printer is turned on and connected but is not ready to open a session. If there is no indicatation that the printer even exists, then the code EXIT_PRNERR_NOT_RESPONDING is more appropriate. For example, a TCP connect attempt results in "connection refused", this code is probably appropriate. If however no response at all is received (i.e., the connexion attempt times out), then EXIT_PRNERR_NOT_RESPONDING is more appropriate.

An interface should not call `alert()` if it intends to exit with this code. An engaged printer isn't considered a condition worthy of notation in the printer's alerts log.

EXIT_STARVED

If the interface program cannot perform its function because a it cannot obtain a sufficient quantity of a finite system resource such as RAM or file descriptors, it should exit with this code. It should not call `alert()` since the problem is not related to the printer. Since the condition is presumably temporary, the operation will be retried after a short delay.

EXIT_PRNERR_NORETRY_ACCESS_DENIED

If when the interace program attempts to connect to the printer it receives unambiguous notification that the connection is refused on the basis of of access control rules or failure to sucessfully complete an authenticaition process, then it should call `alert()` to append a suitable message to the printers alert log and then exit with this code. Since correcting this condition will require administrator intervention (either by reconfiguring the printer itself to allow access reconfiguration the print queue to supply the necessary credentials), the operation will not be retried. Once the problem has been corrected, the command **ppop start** must be used to restart the printer queue.

EXIT_PRNERR_NOT_RESPONDING

If the printer did not respond to the connexion attempt in any way, then the interface program should call `alert()` in order to append an appropriate message to the printers alerts log and then exit with this code. The printer will be show to be in an error state and the operation will be retried after a delay.

If the printer responded indicating that it was unready or unwilling to accept the connection, then a different error code such as EXIT_PRNERR_ENGAGED or EXIT_PRNERR_NORETRY_ACCESS_DENIED should be used instead.

EXIT_PRNERR_NORETRY_BAD_SETTINGS

If the interfaces finds that any of parameters two through six are syntactically incorrect, or logically incompatible, or have values which are incompatible with the interface program or the means of communication with the printer, it should call `alert()` to append a message to the printers alerts log and then exit with this code. Printing will be halted until an administrator changes the settings and restarts the printer with the **ppop start** command.

The interface should not use this code to report that an address lookup failed. The code EXIT_PRNERR_NO_SUCH_ADDRESS should be used instead. The only address problem that should be reported with this code is an unparsable address.

EXIT_PRNERR_NO_SUCH_ADDRESS

This exit code should be used if an attempt to look up the specified printer address fails. This code should be used if no answer was received, the failure might be temporary, or the failure may indicate that the device is simply turned off. An appropriate message should be posted to the printers alerts log by calling `alert()`. The operation will be retried after a delay.

If the result is definite information that a printer with the indicated address does not exist, then EXIT_PRNERR_NORETRY_NO_SUCH_ADDRESS should be used instead.

EXIT_PRNERR_NORETRY_NO_SUCH_ADDRESS

This code should be used when printer address lookup results in an answer which unambiguously indicates that the address does not exist. An appropriate message should be posted to the printers alerts log by calling `alert()`. The operation will not be retried until an operator intervens by running the **ppop start** command.

The interface program should not return any value other than those defined above. Any undefined value will be interpreted as a EXIT_PRNERR. Note that Perl's `die` returns the code 255, which is not among those defined above.

Posting Alerts

An interface program can post alerts to the printer's alerts log by calling the alert() function or runing the **alert** command. An interface program is required to do so before exiting with any of the codes whose names begin with EXIT_PRNERR.

The function `alert()` requires three or more parameters. The first is the name of the printer (from `argv[1]`). The second is a boolean. One may build up the alert message by calling `alert()` several times to add lines to it. This boolean should be TRUE on the first call and FALSE on subsequent calls. The third parameter is a `printf()` style format string. Any addition parameters are values for interpolation into the format string.

A newline will automatically be appended to the format string. Format strings containing embedded newlines are an acceptable way to create multi−line alert messages.

Here is a simplified example of how one might do this in C. It is somewhat contrived. It verifies the address by making sure that it begins with "/dev/".

```
if(strncmp(argv[2], "/dev/", 5) != 0)
        {
        alert(argv[1], TRUE, "Address \"%s\" is syntactically invalid.", argv[2]);
        exit(EXIT_PRNERR_BAD_SETTINGS);
        }
```

For interfaces written as shell scripts, the `alert()` has been wrapped in a tiny program. It may be used like this:

```
. lib/interface.sh

if [ "`echo $2 | cut −c1−5`" != "/dev/" ]
        then
        lib/alert $1 TRUE "The address \"$2\" is syntactically invalid."
        exit(EXIT_PRNERR_BAD_SETTINGS);
        fi
```

Robust Bidirectional Operation

If your interface program supports bidirectional communication with the printer (which is refered to elsewhere as "feedback"), care should be taken to avoid situations which could cause communications to lock up. Lockups can occur if your interface program fails to give top priority to receiving messages from the printer. If the printer has a large amount of data to transmit to the interface program, its output buffer could fill. When a printer has a message to send, it will not process additional input until it has placed the message in its output buffer. If your interface program refuses to accept data from the printer until after the printer has accepted the next data block, a lockup will occur and both parties will wait forever. A particularly insidious aspect of this problem is that it will not happen every time. It is most likely to happen when the printer has a great deal of data to send back, such as query results or status messages.

This means that your interface program must place the file descriptor connected to the printer in non−blocking mode. When activity is detected (perhaps by the `select()` function), data that can be read from the desciptor must be read. It should then be sent out on stdout. Since **pprdrv** lives by the same rules, giving highest priority to the data which your interface program sends to its stdout, it is not absolutely necessary to run stdout in non−blocking mode.

Your interface program must not block on reads from stdin. This is because, after the job has been transmitted, **pprdrv** tries to keep the connexion open until the job is completed. It will stop sending data. But your interface program must go on relaying messages from the printer to **pprdrv**. These messages are used by **pprdrv** not only to keep the printer status up−to−date, but also to determine when the job has been completed. If your interface program blocks on stdin, then PPR will likely get stuck at the end of each job.

The interfaces **tcpip** and **serial** are good sources of example code. They share a function call `int_copy_job()`. This function handles correct two−way communication between stdin, stdout, and the printer file descriptor.

In the same directory you will find several PostScript files with names such as `feedback_test1.ps`. If you print them using your interface program, then will do things such as commanding the printer to send huge amounts of data back. They will quickly smoke out interfaces with fragile two−way communication implementions.

Special Messages

There are special messages which an interface program can send to **pprdrv** to inform it of its progress or to report the status of the printer. These messages are similiar to those which many PostScript printers send back to the computer connecting to them.

Note that an interface program is not required to send any of these messages. However, their use may result in better user feedback.

%%[PPR address lookup]%%

This message indicates that the interface program has parsed its arguments without finding any problems and is not begining a potentially time−consuming address lookup. This message is simply ignored by **pprdrv**, but when **ppad** is running the interface in probe mode it will extend the timeout after receiving this message.

%%[PPR connecting]%%

This message indicates that the interface is begining a the potentially time−consuming process of connecting to the printer. When **pprdrv** receives this message, it sets the cooresponding flag in a status file so that **ppop status** will show the user that a connexion attempt is in progress.

%%[PPR connected]%%

This message indicate that the connexion process has been complete sucessfully.

%%[PrinterError: `message`]%%

Indicates that the printer is unable to print due some disabling condition described by *`message`*. If it is among those listed in `lw−messages.conf`, then **pprdrv** will alter the printer status appropriately.

%%[status: `message`]%%

Indicates a printer condition that is either normal, transient, not critical, or not yet critical. If it is among those listed in `lw−messages.conf`, then **pprdrv** will alter the printer status appropriately.

%%[PrinterError: out of paper]%%

Indicates that the printer has gone off line because it is out of paper.

%%[PrinterError: off line]%%

Indicates that the printer explicitly indicates that it has been taken off line, generally by a user pressing a button. If the interface prints this message before exiting with the code EXIT_ENGAGED, then **ppop status** will show the printer status as "off line" rather than "otherwise engaged or off line".

%%[PrinterError: printer disconnected or powered down]%%

Indicates that a printer directly connected to the server does not appear to be electrically alive at the end of its cable.

%%[status: busy]%%

Indicates that the printer is online and ready and that any disabling PrinterError conditions (such as off line) have cleared.

%%[PPR SNMP: `hrDeviceStatus` `hrPrinterStatus` `hrPrinterDetectedErrorState` *]%%*

This message reports the SNMP status of the printer. The value `hrDeviceStatus` and `hrPrinterStatus` are decimal integers. The value `hrPrinterDetectedErrorState` is an eight digit hexadecimal unsigned integer.

Probe Mode

The **ppad ppdq** and **ppad ppd query** commands provide a way to automatically determine a printer's type. They use a number of techniques to accomplish this goal.

One technique is to connect to the printer and sending it query messages to which it will hopefully respond. These queries use langauges such as PostScript and PJL. These techniques do not generally change according to the connection method, so interface programs need only support two−way communication in order to support them.

But some connection methods, such as USB and IP may provide additional methods for obtaining information about the printer, such as alternative communications channels. For example, many printers which accept jobs over TCP/IP also response to SNMP queries. Since these methods depend very much on the method of connecting to the printer, probes of this type, called out−of−band probes, are implemented in interface programs.

Interface programs which support out−of−band probe will recognize and act on `--probe`. It will interrogate the printer in some way rather than connecting to it with the intention of transmitting a print job. The results of the probe are sent back to **pprdrv** as a series of `PROBE:` lines.

If the probe may take more than a few seconds, it is recommended that the interface immedately send a `PROBE:` line with no value in order to let **ppad** know that it supports probe mode. Otherwise, **ppad** may conclude that the interface program doesn't implement `--probe` and give up too soon.

If the interface is able to obtain information about the printer, it should report it by printing addional `PROBE:` lines. These show be in the form `PROBE:` `name=value`. The `value` should not be quoted, even if it contains spaces. Currently defined names are listed below. Since the probe feature is very new, this list is subject to change.

PostScript Product=

The PostScript product name. The value should be the bare name without surounding partheses.

PostScript Version=
The PostScript interpreter's version number.

PostScript Revision=
The PostScript interpreter's revision number.

SNMP sysDescr=
???

SNMP hrDeviceDescr=
???

1284DeviceID MANUFACTURER=
???

1284DeviceID MFG=
same as above

1284DeviceID MODEL=
???

1284DeviceID MDL=
same as above

When running in probe mode, the interface program will be invoked with the printer name on the command line set to "−". The `alert()` knows about this special name. When it is asked to post an alert to the printer "−", it sends the alert to stderr instead. The result is that the messages will be visible to the user of **ppad** rather than being hidden away in the alert log.

Input Filters

If **ppr** determines that the input file is not PostScript, it will seek to use a filter to convert it to PostScript. This appendix provides the information you will need to write your own PPR input filters.

The filters are found in the directory `/usr/lib/ppr/filters/`. Each of these files has a name that consists of `filter_` followed by the PPR input type name. For example, the filter for JPEG files is called `/usr/lib/ppr/filters/filter_jpeg`.

A filter should read the file from STDIN and write PostScript code on STDOUT. If it must, it can write messages on STDERR. STDIN is guaranteed to be seekable. Messages sent to STDERR will go wherever STDERR was going when **ppr** was invoked.

The parameters are as follows:

1. The first parameter is the list of filter options. These are expressed as a space seperated list of name−value pairs. The name and value are joined by an equal sign. The options list is formed by concatrnating the contents of the `DefFiltOpts:` line in the printer or group configuration file with the contents of any `−o` lines the user put on the **ppr** command line. Before passing the option list to the filter, **ppr** pre−processes it. Any options whose names begin with a file type name and a hyphen

will have the file type name and the hyphen removed if the file type name matches the filter being invoked, otherwise, such options are discarded. The names of the parameters (the part to the left of the equals sign) are converted to lower case. The values (the part to the right of the equals sign) are not. A responder should ignore any option it does not recognize. If the same option appears more than once, the value from the last instance is the one that should be used.

2. The name of the printer or group to which the job was submitted. This will generally be ignored.

3. The third is the job title. This may be used by filters which format their input as pages with headers and footers.

4. The directory which was current when **ppr** was invoked. This is used by the TeX, TeXinfo, and DVI filters when searching for include files.

When a filter is invoked, the environment variable `IFS` is set to a space and a tab, and the variable `PATH` is set to a value which is just adequate to find standard shell script helper programs such as test, sed, and grep. On most systems, that value of `PATH` is `/bin:/usr/bin`.

Here is an example. Suppose this line is in the printer's configuration file:

```
DefFiltOpts: level=2 colour=False resolution=300 freevm=1048576 mfmode=CanonCX
```

and the user submits a JPEG (JFIF) file with this command:

```
$ ppr −d myprn −o noisy=no −o 'fortran−width=130 jpeg−noisy=yes' picture.jpg
```

The filter will be invoked like this:

```
filter_jpeg 'level=2 colour=False resolution=300 freevm=1048576 mfmode=CanonCX noisy=no noisy=yes
```

A filter should interpret any options it recognizes and ignore any it does not. If it finds two contradictory options, it should obey the last one. In the example above, the option `noisy=yes` is the one that prevails. The options `freevm=1048576` and `mfmodes=CanonCX` would be ignored simply because the JPEG filter has no code to use them.

If the filter exits with a value other than 0, the job will not be discarded. A message may be informed by printing on stderr or invoking a responder, the exect behaviour being controled by the `−e` switch.

When a filter is executed, the real user id is that of the user who executed **ppr**. The effective user id and the saved user id's are `ppr`. The real group id is the same as it was when **ppr** was executed. The effective and saved group id's are `ppop`.

It is possible to determine precisely what filter is being executed with what arguments by running **ppr** with the `−G infile:filter` option.

The filters supplied with PPR, together with their options are described in the ppr(1) man page, under the section for the `−T` switch.

Fonts

Font Substitution Configuration File

This file resides in the directory `/usr/lib/ppr/lib/`. It is replaced whenever a new version of PPR is installed, so if you modify it you should keep a copy of your modifications elsewhere.

The file contains a list of PostScript font names and possible substitute fonts.

Any line that has # or `;` in the first column is a comment. Blank lines are ignored. A font substitution record begins with the name of the font for which there are substitutes. The name should start in column one and be on a line by itself. The list of possible substitutes follows, one per line. Each substitute font line should start with a space or tab. You may list as many substitute fonts as you like. The first substitute font to be found in the printer's PPD file, in the cache directories or in the index created with **ppr−index fonts** will be used.

Here is an example of four records which indicate that the IBM Courier fonts are suitable substitutes for the Adobe Courier fonts:

```
Courier
        IBMCourier
Courier−Bold
        IBMCourier−Bold
Courier−Oblique
        IBMCourier−Italic
Courier−BoldOblique
        IBMCourier−BoldItalic
```

You have the option of specifying a PostScript transform matrix to be applied to the substitute font. Generally this will be used to adjust the width. The should appear on the substitute font line, after the substitute font name. Here is an example:

```
Helvetica−Condensed
        Helvetica [0.80 0 0 1 0 0]
Helvetica−Condensed−Bold
        Helvetica−Bold [0.80 0 0 1 0 0]
Helvetica−Condensed−Oblique
        Helvetica−Oblique [0.80 0 0 1 0 0]
Helvetica−Condensed−BoldOblique
        Helvetica−BoldOblique [0.80 0 0 1 0 0]
```

This approximates Helvetica Condensed by scaling Helvetica to 80% of its normal width. The role of the other members of the matrix, refer to the PostScript language reference manual.

MetaFont Modes Configuration File

This file `mfmodes.conf` is used by **ppad** when setting the default filter options. Specifically, the `mfmode=` option is selected with the aid of this file. The `mfmode=` option is used by the DVI filter to select an appropriate MetaFont mode for a given printer.

Before consulting this file, **ppad** reads the printer's PPD file and extracts the values from the following lines:

```
*Product:
*ModelName:
```

```
*NickName:
*DefaultResolution:
```

The extracted values are then compared to values on lines in the `mfmodes.conf` file. Each line in the `mfmodes.conf` file has the following format:

```
product:modelname:nickname:resolution:mfmode
```

The file is read top to bottom until a match is found or the end is reached. The information from the PPD file is compared to the first four fields of each line. A `*` may be used as a wildcard in any or all of the first four fields. When a match is found, the value from the fifth field is used as the value for the default filter option *mfmode=*.

Identification by Product

The `*Product:` line from the PPD file generally identifies the manufacturer and model line of which the printer is a part. Since all printers which use one product string generally use the same print mechanism, this parameter alone is usually enough to make selection of the correct MetaFont mode possible. For this reason, most entries in the `mfmodes.conf` file will have the *`product`* field filled in but the *`modelname`*, *`nickname`*, and *`resolution`* fields will all be `*`.

For example, the following line:

```
LaserJet 4:*:*:*:ljfour
```

will match if the `*Product:` line from the PPD file has a value of `(LaserJet 4)`. The fact that fields two through four contain astrisks indicates that any value is acceptable for product, modelname, and resolution.

Identification by ModelName

The GhostScript interpreter has a product string of `Ghostscript` or `Alladin GhostScript`. Therefor, the printer must be identified by means of its PPD files `*ModelName:` line. Here are some reasonable configuration lines for printers driven by Ghostscript:

```
*:Dot Matrix 24 pin Ghostscript:*:*:NEChi
*:HP LaserJet III Ghostscript:*:*:CanonCX
*:HP DeskJet 500 Ghostscript:*:*:HPDeskJet
```

Identification by NickName

In a PPD file obtained from Adobe or the printer's manufacture, the `*NickName:` line is identical to the `*ModelName:` line. You might change it if you make a special hacked−up copy of the PPD file for a particular printer. For example, you might change it to `David's HP DeskJet 500 Ghostscript`.

There are valid reasons for using a modified PPD file, but why changes to the PPD file should dictate a different MetaFont mode is hard to say. (Changes to the `*DefaultResolution:` line are covered by the next section.) Unless you know a good reason not to, you should always put a "*" in this field.

Identification by Resolution

Generally, you can just put `*` in the *`resolution`* field. There are however two possible reasons for filling in the value from the PPD file's `*DefaultResolution:` line.

One is if the printer's resolution can be changed. A change in resolution requires a change in the MetaFont mode. You might have several different PPD files for the same make and model of printer, one for each resolution. Here is a (fictitous) example:

```
*:HP LaserJet III Ghostscript:*:300dpi:CanonCX
*:HP LaserJet III Ghostscript 150DPI:*:150dpi:ljlo
```

The other reason for putting a value other than `*` in the *`resolution`* field is if the line is one at the end of the file which is intened to to be a best guess for printers which have not matched any of the lines above. These are some reasonable last resort lines:

```
*:*:*:300dpi:CanonCX
*:*:*:600dpi:ljfour
*:*:*:360dpi:NEChi
```

Selecting MetaFont Mode Names

The MetaFont mode names in the `mfmodes.conf` file that comes with PPR are taken from the `modes.mf` file maintained by Karl Berry. A recent version of his `modes.mf` file is distributed with the PPR source code, in the `misc` directory. His file defines a number of aliases for each mode. When adding entries to PPR's `mfmodes.conf` file you should try not to use two different names that both refer to the same mode in `modes.mf` because that would result in the generation of duplicate sets of identical pk font files. The `mfmodes.conf` file supplied with PPR always uses the first alias from Karl Berry's `modes.mf` file which may be abtained from ftp://ftp.tug.org/tex/modes.mf.

Protocol for Comunicating with pprd

This section describes the protocol which use commands such as **ppr**, **ppop**, and **ppad** use to communicate with **pprd**. This information may assist those attempting to understand the source code. However, one should not implement this protocol since it changes from version to version. Instead one should run **ppop** with the `-M` option and parse its output. For this reason, only a few example commands are described here.

The spooler daemon, **pprd** receives commands over a named pipe and acts on them. The program **ppr** sends one such command to **pprd** in order to inform it that a new jobs has been placed in the queue directories. The program **ppad** uses two different commands which it uses to inform **pprd** that a printer, group configuration file has been modified. Finally, **ppop** has many commands which it uses to control **pprd** or to request information from it.

Accepting Jobs

Once the job submission program **ppr** has created the queue files it sends a command to **pprd** telling it that the job is there and is ready for printing or transfer to the remote system. This command takes the form:

```
j destination_node destination_queue id subid home_node initial_priority
```

The *`destination_node`* is the node to which the job should be sent. Since transmission to remote notes is not yet supported, this will always be the name of the local node. The *`destination_queue`* is the name of the print queue on the destination node. The *`id`* and *`subid`* are the queue id and the job fragment number respectively. The *`subid`* will most often be zero. The *`home_node`* is the node name of the system on which the job originated (again, the local node).

Reloading Configuration Files

Whenever a printer or group configuration file is modified or a new one created, **pprd** must be directed to read it. To inform **pprd** that it must read a new or revised printer configuration file, a command line of this form is sent:

```
NP printername
```

To indicate that a group configuration file must be re−read or that a new group configuration file must be read for the first time:

```
NG groupname
```

Neither command is acknowledged by **pprd**.

Yielding Information to ppop

The utility **ppop** must communicate with **pprd** in order to list the queue, show the status of printers, mount forms, and do other tasks. Most of these communications take the form of a query and a reply.

As currently implemented, the interprocess communications is crude but effective. The queries are sent by writing to **pprd**'s named pipe. The replies are sent back in temporary files. The command which **ppop** writes to the name pipe begin with **ppop**'s process id. After writing the command, **ppop** waits for a signal, USR1 to be specific. The process ID is read by **ppad** which creates a temporary file called `/tmp/ppr−ppop−`*`pid`* where *`pid`* is the process id sent by **ppop**. Once **pprd** has finished writing the reply, it closes the temporary file and sends SIGUSR1 to **ppop** which opens the file, deletes it, reads its contents, and formats and displays said contents.

Many of the replies take the form of a numberic exit code for **ppop** and a human−language message to display. Some include many additional lines of results but only if the code in the first line is zero (indicating success).

Much of the code in the parts of **ppop** and **pprd** which communication with one another was written before any attempt was mode to internationalize PPR. This is a problem because **pprd** may be generating messages in one language while **ppop** is generating them in another. The plan is to eventually eliminate all user visible messages from these files leaving only code information. At the same time **ppop** will be modified to digest this information and produce text in the users language.

How Continuous Queue and Printer Status Display Works

[This section has not been written yet.]

Requirements for a Responder

A responder is a small program. All responders are stored in the responders directory `/usr/ppr/responders`. The responder which will eventually be used to tell the user what happened to the job is selected when the job is submitted. It is selected using **ppr**'s **−m**. switch. The argument to the **−m** switch is the name of the program in the responders directory which should be used. A responder can be very simple. Here is an example:

```
#!/bin/sh
echo "Message for $1:\n$3" | write $2
exit 0
```

The responder shown above is a stript down version of the responder **write**. If the **−m** switch is not used then the name of the responder will be read from the environment variable PPR_RESPONDER. If PPR_RESPONDER is not defined then the responder **write** will be used.

A responder program is invoked with the name of the user as the first parameter. It is the name just as it appears in queue listings and on banner pages.

The address to which the message should be sent is the second parameter. The address is specified at the time **ppr** is invoked by using the **−r** switch. The proper format for the address depends on the responder. If the **−r** switch is absent then the value of the environment variable PPR_RESPONDER_ADDRESS is used. If that too is absent then the name of the Unix user who invoked **ppr** is used. The value used in the absence of both **−r** and PPR_RESPONDER_ADDRESS is appropriate since the default responder is a script which invokes the Unix program **write**.

The third parameter is the suggested message text. The message text will contain embedded line feeds. The length of the lines will depend on the responder. The routine get_responder_width() in `libppr/reswidth.c` determines what length to limit the lines to. For most responders, this routine returns 0 which means to use a predetermined set of line breaks which generally results in lines less than 80 characters long.

The fourth parameter is reserved for future use, at present it is an empty string.

The fifth parameter is a space separated list of responder options. Each of this options is a name=value pair. Common options include **printed=no** and **timeout=60**. The value of this option comes from the **ppr −−responder−options** switch or from the environment variable PPR_RESPONDER_OPTIONS. A responder should ignore any options it does not recognize.

The sixth parameter is a code number which represents the approximate content of the message. It is provided in case a responder wants to send different types of message by different methods or wishes to provide its own wording for one or more of the messages. For example, it might want to send a popup message but then follow it up with email if the message indicated that the job was arrested. The code numbers are defined in C include file `include/respond.h`, the Bourne shell script include file `lib/respond.sh`, and the Perl include file `lib/respond.ph`.

The seventh parameter contains the complete job id. This is probably only useful if the responder is constructing its own messages since the default messages provided in the third parameter already mention the job id. The job id is not provided in the normal form since that is rather difficult to parse. In stead, the elements are separated by spaces. The elements are, in order, destination node, *`destination name`*, *`queue id number`*, *`sub id number`*, and *`home node`*. For example, `mouse chipmunk 1000 0` `mouse`. This id would normally appear as `mouse:chipmunk−1000.0(mouse)` or more likely in abreviated form as `chipmunk−1000`.

The eight parameter is also necessary for constructing replacement messages. Some of the messages normally contain a blank space which is filled with a piece of text. Normally this is the name of the printer which the job was printed on, but if the user database is being used and PPR refuses access the the printer, the piece of text is the name of the user who was refused access. At other times the piece of text will be an error message.

The ninth parameter is the title of the job. This title will be derived from a "%%Title:" line if present. Failing that, the title will be the name of the file being printed. If the file was received on stdin, this field will generally be blank.

The tenth parameter is the time at which the job was submitted. It is represented in Unix format (as a count of the seconds since 12:00am, January 1, 1970). The program **lib/time_elapsed** may be used to express this time in terms of how far in the past it is.

The eleventh parameter will contain a short message which indicates the reason the job was arrested. If the response code does not indicated that the job has been arrested, this string will be blank.

The twelth parameter contains the number of pages printed. If this is unknown, it is "?". If the response code (the fourth parameter) has a value other than **RESP_FINISHED** then the contents of this field is undefined.

When the responder is run, stdin with either be connected to `/dev/null` or it will be connected to the job's log file. This is so that the responder may send the job log back to the user. Some of the supplied responders exploit this feature.

When exiting, the responder should return a value of zero if the message was delivered. It should also return a value of zero if the addresse was not found. Non−zero exit values should be reserved for truly abnormal conditions such as insufficient system resources or syntactically invalid addresses. If a responder does return a non−zero exit value, **ppr** will print a notice to that effect on stderr or **pprd** will put a notice in its log file, depending on which one invoked the responder.

When the responder is invoked by **pprd**, it will always have a real uid, effective uid and saved uid of `ppr`. When it is invoked by **ppr** (due to the use of the **−e** responder option) things will be different. The program **ppr** is setuid `ppr`, so the effective and saved uids are `ppr` and the real uid is the id of the user who invoked it. Just before executing the responder **ppr** sets the effective user id equal to the real user id. This last feature is not intended as a security measure, rather, the **xwin** responder will not work if this is not done since the X library uses access() on the .Xauthority file before trying to open it which means that the .Xauthority file in the user's home directory must be readable under both the real and the effective uids. This feature may be overridden by setting the setuid bit on the responder. If this is done then the effective uid will remain `ppr`.

Custom Hooks

A custom hooks is a small program which PPR runs at predetermined points in the process of sending a job to a printer. Anything that the program sends to standard output will be transmitted to the printer. The program has access to the queue file, so it can determine the characteristics of the job. For example, you could write a custom hook program which printed a banner page. If a print queue is properly configured, your program will be used to print banner pages instead of PPR's internal banner page printing code.

In the file `pprdrv/pprdrv.h` the following constants are defined:

```
#define CUSTOM_HOOK_BANNER 1
#define CUSTOM_HOOK_TRAILER 2
#define CUSTOM_HOOK_COMMENTS 4
#define CUSTOM_HOOK_PROLOG 8
#define CUSTOM_HOOK_DOCSETUP 16
#define CUSTOM_HOOK_PAGEZERO 32
```

We will get to what they individually mean in a minute. For now let us say that they each one of them stands for a point at which your custom hook may be called to insert additional text into the PostScript job. Notice that they are powers of two. That means that they may be added up to produce a number that represents a set of choices as a bitmap. In order to tell PPR at which insertion points in the job a custom hook should be run, one adds up the code numbers which represent the desired points.

Where does this value go? You should enter it, together with the name of your custom hook program, on a line in the printer configuration file. The line has the following format:

```
CustomHook: bitmask program
```

The *`bitmask`* is the total of the codes and the *`program`* is the name of your custom hook program. You should probably specify the complete path. Since **ppad** doesn't have a command for manipulating entries like this one, you should add the line using a text editor. You will find the file in `/etc/ppr/printers/`.

When your custom hook program is run, stdout will be connected to the printer. That is, anything you print on stdout will be transmitted to the printer. What you are expected to send to the printer depends on the insertion point and will be discussed below when the insertin points are described. Stdin will be connected to `/dev/null`. Stderr will be directed to the **pprdrv** log file (`/var/spool/ppr/logs/pprdrv`). While developing a custom hook program it is helpful to print debugging messages to stderr.

The custom hook program will receive three command line parameters. The first parameter is the code for the part of the PostScript document that is being generated. The second parameter gives details. For most document parts it is zero. The third parameter is the full queue ID of the job. It may be passed to **ppop qquery** to get details about the job or, since it is also the name of the queue file, it may be used to open the queue file in `/var/spool/ppr/queue/`.

And now for a description of the various points at which your custom hook program can be called.

CUSTOM_HOOK_BANNER (1)

> The custom hook program is invoked to print a substitute banner page. The regular banner page is suppressed. Remember that a banner page comes before the job in the sense that it is placed in the output tray in front of page one. Whether it is actually printed before the job depends on whether PPR thinks the tray is face up or face down. If your code needs to know if is being printed (chronologically) before the job's pages or after, it should examine its second command line parameter which will be `0` for before the job and `1` for after. [Note: verify that I have that right.] Note that your custom hook program should generate a complete PostScript document. The printer's PostScript interpreter will be reset before and after your banner page.

CUSTOM_HOOK_TRAILER (2)

> The custom hook program is invoked to print a substitute trailer page. Again, the second parameter indicates whether the body of the job has been printed yet.

CUSTOM_HOOK_COMMENTS (4)

The custom hook program is invoked just before the document header comments are sent. Thus the program can add any additional document header comments it likes. Notice that each and every line generated must begin with `%` and be followed by a printable character other than space tab or newline.

CUSTOM_HOOK_PROLOG (8)

The custom hook program is invoked at the end of the document prolog maybe so that it can insert any procedure sets it might need. It is unlikely you would want to do this, unless perhaps the intent of your custom hook is to do something like N−Up printing. (Of course, PPR can already do N−Up printing.)

CUSTOM_HOOK_DOCSETUP (16)

The custom hook program is invoked at the end of the document setup section. This is where you would turn on your replacement N−Up printing implementation.

CUSTOM_HOOK_PAGEZERO (32)

The custom hook program is invoked just before the first page of the document. It can insert one or more additional pages. Basically, it can be used to add a banner page inside a job. This is for use in environments where banner pages outside the job would confuse some system furthure down the line. This is the case when printing to a Xerox Docutech. Since the document setup section's code may have already set up transform matrixes which could squish or shift the page you will be generating, it should wrap itself in `save initmatrix` *...* `restore`.

You may have noticed that there is no provision for calling separate custom hook programs for separate insertion points. Your custom hook program should examine its first command line parameter and select the code path that cooresponds to the current insertion point. If you really need separate program, you will have to create a shell custom hook program that examines its first parameter and then executes the desired program.

If you need to invoke any printer−specific features, such as to select a particular paper size, you should print a DSC comment line like this one:

```
%%%%IncludeFeature: feature setting
```

The *`features`* is a PPD feature name such as `*PageSize` and the value is on of the possible values such as `Letter` or `A4`. Before your program's output is sent to the printer, this comment will be replaced with the proper code from the printer's PPD file.

Of course, since you will be generating PostScript code or at least PostScript comments, the is always the possibility that you will turn a job into an invalid PostScript program that won't print. In that case, PPR will come to your rescue by arresting the job so that you can examine its log with **ppop log** in order to see what the PostScript error messages are.

You will almost certainly want to look at the final output sent to the printer in order to verify that your text is being inserted where you expect. To do that, set up a test queue using the **dummy** interface. It will print to a file which you can then examine.

PPR currently includes one sample custom hook program. It is installed at `/usr/lib/ppr/lib/custom_hook_docutech`. Among other things it demonstrates how to do CUSTOM_HOOK_PAGEZERO.