Impressario™ Programming Guide

CONTRIBUTORS

Written by David Graves

Updated by Don Moccia and Martha Levine

Edited by Nancy Schweiger and Christina Cary

Production by Kirsten Johnson

Engineering contributions by Roger Chickering, Ken Kershner, Baron Roberts,
    David Story, and Craig Upson

Engineering update by Ray Niblett

St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower image
    courtesy of Xavier Berenguer, Animatica.

Impressario™ Programming Guide
Document Number 007-1633-050

Contents

List of Figures

List of Tables

About This Guide

The *Impressario Programming Guide* is written for the following users:

- Printer driver developers

- Scanner driver developers

- Application program developers who need to print or scan from their applications

About This Guide

Impressario™ is a printing and scanning environment for Silicon Graphics® IRIS® workstations. The Developer's Kit, included with Impressario, provides solutions for a wide range of UNIX® audiences: printer driver and scanner driver developers, application program developers, and end users.

The goal of Impressario is to provide an intuitive, friendly, and reliable interface for end users, while increasing system capability and performance for driver and application developers. Users can simply drag and drop a file onto a printer icon to print the file. Other graphical tools in the end user's environment provide information on the capabilities and status of any accessible printer or scanner.

The Impressario printing environment provides two main end-user enhancements:

- support for a wide range of printers, from high-quality color printers to high-speed black-and-white printers

- graphical printing tools that allow a user to submit a print job and monitor the status of the job and the printer

Impressario enables developers of printer drivers and scanner drivers to showcase each product's special features and capabilities and present them to the user via a graphical dialog box. Application programmers can greatly reduce the development time required to support printing and scanning functions.

The manual applies to all releases of Impressario 2. Please review the release notes for all releases of Impressario

Tip: The Impressario release notes contain the most recent information about the product. They are provided online and can be read using *relnotes* or *grelnotes*. In addition, the directory */usr/impressario* contains information of interest to both application developers and end users.

The Impressario *printing environment* is built on top of the AT&T System V®, Release 3 (SVR3) printer spooling interface. Model files, filters, and printer drivers are provided to convert a wide variety of file types (ISO text files, Silicon Graphics image files, PostScript™ files, and so forth) to formats for both raster printers and PostScript printers. Using the Impressario host-based PostScript interpreter, it is possible to print PostScript documents to raster printers with performance that greatly surpasses printer-based PostScript interpreters. Impressario also includes the PrintBox Motif widget™, a graphical user interface (GUI) for printing.

Impressario *server software* contains filters and drivers for sending jobs from a host workstation to a printer. In addition, all Impressario printer drivers maintain status information in a globally available printer object database (POD).

The Impressario *scanning environment* provides generic scanner support. Impressario scanner application programs and Impressario scanner drivers run as separate executables, enabling any scanning application to interact with all scanner drivers.

Impressario gives application developers a number of valuable resources, including

- *libspool*, a printer spooling system abstraction library that enables complex printing functions to be defined with only a few lines of code

- application programming interfaces for easy access to the end user's scanning environment

- a network-transparent version of the printer object database library routines to inquire directly about printer configuration and status

Audience

The *Impressario Programming Guide* is written for the following users:

- printer driver developers

- scanner driver developers

- application program developers who need to print or scan from their applications

New Features

The 2.*n* release of Impressario contains these new features:

An Adobe Level II Configurable PostScript Interpreter (CPSI)
> The CPSI supports banded devices, such as inkjet plotters and inkjet printers, reducing the amount of host memory needed to use them.

A Generic Color PostScript driver
> This driver converts files to be printed into PostScript, which it sends to printers with PostScript interpreters, including selected color printers.

A new filter, text2ps, that converts ASCII text to PostScript
> All Impressario drivers now use text2ps to convert ASCII text files.

The Developer's Kit
> It is bundled with Impressario and is no longer a separate option.

Impressario 2.0 also adds support for the following:

- Hewlett-Packard DesignJet® 750C plotters

- Hewlett-Packard DeskJet® 660C, 850C, 855C, and 1600C color inkjet printers

- Hewlett-Packard LaserJet® 4V, 4 Plus, 4Si, 5L, 5P, and 5Si printers

- Hewlett-Packard JetDirect™ network adaptors for various printers

- Lexmark Optra® R, Rx, L, Lx, and Lxi monochrome laser printers

- Epson Color Stylus™, Stylus Pro™, and Stylus Pro XL printers

- International Color Consortium (ICC) color profiles for the color management of image files, such as GIF™ and TIFF™

Note: The release notes list all supported printers and scanners, and have details about the JetDirect network adaptors. Additional drivers will be added for releases following Impressario 2.0 (i.e. 2.1, 2.2, etc).

How to Use This Guide

Since this guide has four separate audiences, the list provided below gives the areas of most interest to each user:

- Printer driver developers should read chapters 1–6, 10, 11, and 12.
- Printer application program developers should read chapters 1, 6, and 12.
- Scanner driver developers should read chapters 1, 2, 7–9, 11, and 12.
- Scanner application program developers should read chapters 1, 2, 9, and 12.

Conventions Used in This Guide

The followings conventions are intended to help make information easy to access and understand:

italic	Used for arguments in a command line that you replace with a valid value. In text it indicates an argument, button, command, document title, file name, glossary item, new term, or variable. For example:
	Use the *phandler* command to ....
	The *NAME* variable identifies the printer name.
ALL CAPS	Used for defined constants in text. For example:
	The SC_PROGFEED bit should be set.
bold	Denotes command-line options, keywords, and functions. For example:
	The -**o** option directs output.
	SCErrorString() returns a text string.
Courier	Used for code examples and screen displays. For instance, the following is a code example:

```
int
AdvanceFeeder(SCANINFO *scan)
{
    drverr = SCENOFEEDER;
    return -1;
}
```

`Courier bold`	Used for user input. For example:
	`vstiff /usr/tmp/sample.blastfile<Enter>`
< >	Used to enclose arguments, parameters, and nonprinting keys (see above).
[ ]	Enclose optional command arguments. Do not enter the brackets. Example:
	`[optional_entry]`

Document Overview

Chapter 1, "Impressario Architecture," discusses the Impressario printing and scanning architectures and defines Impressario compliance for printer driver and scanner driver developers.

Chapter 2, "Printer Drivers," provides an overview of printer driver processing, plus a detailed analysis and discussion of an example printer driver. The required printer filter/driver options are also covered.

Chapter 3, "Printer Model Files," discusses the printer model files and describes the modifications to be made by printer driver developers to the printer model file template.

Chapter 4, "Printer Graphical Options Panel," discusses the graphical options panel that visually showcases a printer's features. The major topics discussed are options handling, panel layout, development, naming, installation, invocation, and termination.

Chapter 5, "Printing Libraries," describes the libraries used by printer drivers, filters, and applications. The libraries described are the *libspool* library, the *libprintui* library, and the *libpod* library.

Chapter 6, "Scanner Drivers," explains how to write a scanner driver. It provides a detailed analysis of the template scanner driver. The major topics are the driver template, header files, data structures, functions, installation, and testing.

Chapter 7, "Scanner-Specific Options," discusses how to implement scanner-specific options for a scanner driver. The major topics are the options program, perspectives, and installation and testing.

Chapter 8, "Generic Scanner Interface," describes a generic interface between a scanner driver and an application program. The major topics are the coordinate system for scanning, data structures, and data type conventions.

Chapter 9, "Testing for Impressario Compatibility," explains how to use the programs that test printing compatibility with the Impressario environment. It explains how to test Impressario printing compatibility, an Impressario printer, and an Impressario printer software installation.

Chapter 10, "Packaging Your Impressario Product," explains how to package the Impressario software product that you have created.

Chapter 11, "Enhancing Impressario With Plug-Ins," explains how to add new features to the Impressario open architecture. The major topics are how the Impressario file conversion pipeline works, how to add a new filetype to Impressario, and how to use an alternate PostScript RIP.

Appendix A, "Stream TIFF Data Format," describes the Stream TIFF file format, the primary interchange file format between printer filters and drivers; and *libstiff*, a C application program interface (API) used to read and write Stream TIFF files. Stream TIFF is also used by gscan to store images in TIFF files and to scan to the screen (in conjunction with vstiff).

Appendix B, "Silicon Graphics Image File Format API," describes *libimp*, the C-language API for reading and writing Silicon Graphics Image format files. The image processing features of *libimp* are also described.

Appendix C, "Printer Object Database (POD) File Formats," defines the file formats for printer configuration, status, and log files in the POD. The major topics are general syntax, input parsing rules for *libpod* files, printer configuration file format, and printer status file format.

Appendix D, "Transition Notes," explains how to migrate from Impressario 1.2 software to Impressario 2.*n*. It also explains how Impressario application developers and filter/driver developers can take advantage of the new features in Impressario 2.*n*.

Appendix E, "Scanner Driver Architecture," describes the architecture of a scanner driver and discusses the template scanner driver, required and optional functions, and queues and multi-threaded scanner drivers.

Appendix F, "Reference Pages," lists all Impressario online reference pages: general interest, printer developers, and scanner developers.

Appendix G, "Color Management In Impressario," discusses using ICC color profiles and PostScript Color Rendering Dictionaries to promote color consistency among devices such as monitors, scanners, and printers.

Appendix H, "Using the Adobe Configurable PostScript Interpreter," discusses legal restrictions on developers who use software licensed by Silicon Graphics from Adobe Systems.

Related Publications

Online Books

The following books, available online through Silicon Graphics, contain information related to Impressario:

- *Programming on Silicon Graphics Systems: An Overview*, Silicon Graphics, Inc.

- *Indigo Magic Desktop Integration Guide*, Silicon Graphics, Inc.

- *IRIX Device Driver Programming Guide*, Silicon Graphics, Inc.

- *IRIX Device Driver Reference Pages*, Silicon Graphics, Inc.

- *IRIX Admin: Peripheral Devices*, Silicon Graphics, Inc.

- *Volume One: Xlib Programming Manual*, O'Reilly & Associates

- *Volume Four: X Toolkit Intrinsics Programming Manual*, O'Reilly & Associates

- *OSF/Motif Programmer's Guide*, Prentice-Hall, Inc.

- *OSF/Motif Programmer's Reference*, Prentice-Hall, Inc.

- *OSF/Motif Style Guide*, Prentice-Hall, Inc.

- *Impressario User's Guide*, Silicon Graphics, Inc.

Online Release Notes

After installing online documentation, you can view the Impressario release notes. If you have a graphics system, select "Release Notes" from the Help Toolchest to display the grelnotes graphical browser. Refer to the grelnotes(1) reference page for information on options to this command. If you do not have a graphics system, you can use the *relnotes* command. Refer to the relnotes(1) reference page for accessing the online release notes.

Adding the product name to the *relnotes* command displays the table of contents for that product's release notes. For example:

```
% relnotes Impressario
  The chapters for the "Impressario" product's release notes are:

  chap title
  1 Introduction
  2 Installation Information
  3 Changes and Additions
  4 Bug Fixes
  5 Known Problems and Workarounds
  6 Documentation Errors

  Use "/usr/sbin/relnotes productname chapter" to view a chapter
```

Online Reference Pages

Appendix F lists the reference pages provided online with Impressario. To access them, enter:

man *page-nam*

Impressario Architecture

This chapter gives an overview of Impressario's printing and scanning architectures. It also explains how to write Impressario compliant printer drivers and scanner drivers. By complying with Impressario guidelines, you make your job easier, ensure a consistent end-user experience, and greatly improve the chances of effortless transitions to future Impressario releases.

Impressario Architecture

This chapter discusses the Impressario printing and scanning architectures and defines Impressario compliance for printer driver and scanner driver developers. By complying with Impressario guidelines, you make your job easier, you ensure a consistent end-user experience, and you greatly improve the chances of effortless transitions to future releases of Impressario.

The following topics are discussed in this chapter:

- "Impressario Printing Architecture" on page 5
- "Compliance for Printer-Driver Developers" on page 6
- "Printing Application Programming Interfaces" on page 8
- "Printing Application Development" on page 12
- "Complying With the Impressario Scanning Architecture" on page 13
- "Developing a Scanner Driver" on page 14
- "Developing a Scanner Application" on page 15

Overview

Impressario allows files of different types to be printed on any installed printer and images to be scanned from a scanner, a workstation monitor, or a Silicon Graphics Image file. A visual end-user environment makes it easy for users to add new devices and for applications to take advantage of those devices by providing graphical interfaces for these purposes:

- installing printers (see the printers(1M) reference page)

- modifying printer settings (see PrintPanel [glp(1)] or *printers*)

- checking printer status (see PrintStatus(1))

- submitting print jobs from applications (see the PrintBox widget)

- installing scanners (see scanners(1M))

- using scanners (see gscan(1))

To maintain a consistent, reliable, and easy-to-use environment, Impressario provides the following libraries for application developers:

libspool	a C application program interface (API) to the UNIX printer spooling system
libprintui	a C graphical user interface (GUI) library for printing that is compatible with Motif
libpod	a C-language API to the Printer Object Database (POD)
libscan	a C-language API to the Impressario scanning system
libstiff	a C-language API for reading and writing Stream TIFF (STIFF) files
libimp	a C-language API for reading and writing Silicon Graphics Image files

The following final, crucial elements must be provided by driver developers:

- For printers: a compliant printer driver that reports printer status through *libpod* and, optionally, a graphical options panel

- For scanners: a scanner driver for the driver side of the generic scanner interface and, optionally, a scanner-specific options panel

Impressario Printing Architecture

This section describes the steps that developers of printer drivers and printing applications must take to comply with Impressario specifications.

Figure 1-1 is an overview of Impressario printing components. A more detailed version of this diagram is available online in */usr/impressario/doc*.
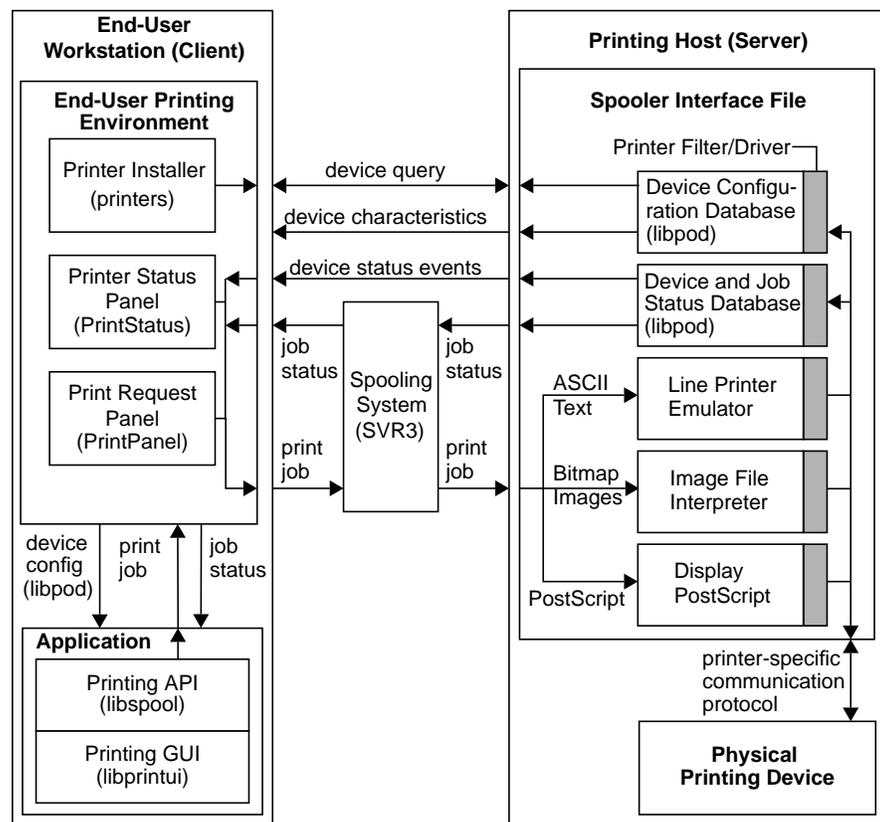
Figure 1-1   Impressario Printing Architecture

The following sections detail the steps that should be followed to achieve Impressario compliance for printer drivers and printing applications.

Compliance for Printer-Driver Developers

The steps shown in Figure 1-2 (discussed below) show how to develop and integrate a printer driver for Impressario.

Step 1	Develop printer driver (required)	See Chapter 3, "Printer Drivers," for detailed instructions.
Step 2	Provide POD files (required)	See Appendix C, "Printer Object Database (POD) File Formats," for detailed information.
Step 3	Create model file (required)	See Chapter 4, "Printer Model Files," for detailed information and instructions.
Step 4	Provide data filters (as needed)	See "Filter/Driver Specification" in Chapter 3 for more information.
Step 5	Create graphical options panel (recommended)	See Chapter 5, "Printer Graphical Options Panel," for detailed information.
Step 6	Package software for distribution (required)	See Chapter 11, "Packaging Your Impressario Product," for detailed information.
Step 7	Verify product on server (required)	See Chapter 10, "Testing for Impressario Compatibility," for detailed information.

Figure 1-2   Printer Driver Development Flowchart

Step 1: Develop Printer Driver (Required)

Your printer driver must comply with the Impressario Filter/Driver Specification (see "The Filter/Driver Specification and psrip" in Chapter 2) so that a model file that is Impressario compliant can execute the driver correctly. This specification describes standard driver behavior and the command-line arguments that must be processed. See */usr/impressario/src/drivers* and Chapter 2 for the source code and an example driver. The driver must update the POD files through calls to "local" functions of the library *libpod*. (See Chapter 5, "Printing Libraries.") The formats of the POD files are described in Appendix C, "Printer Object Database (POD) File Formats."

Step 2: Provide POD Files (Required)

A set of POD files consists of a configuration file, a printer log file, and a printer status file. Each POD file has the same base name as the printer model file. The extensions on these files are: *.config*, *.log*, and *.status*, respectively. To create these files, start with the example set of POD files in the directory */usr/impressario/src/data*. The POD files you create must be installed in the directory */usr/lib/print/data* when you install your software. See Chapter 10, "Packaging Your Impressario Product," for more information.

Step 3: Create Model File (Required)

Your model file must conform to the Impressario model file specification. This is done by starting with the template model file provided with Impressario and adding your developer-specific processing. Model files must be installed in */var/spool/lp/model*. Follow the Impressario model file template to create a model file that properly updates the desktop printer status icon and interacts properly with Impressario subsystems. (See Chapter 3, "Printer Model Files," for more information on model files.)

Step 4: Provide Data Filters (As Needed)

Filters must conform to the filter/driver specification. (See "The Filter/Driver Specification and psrip" in Chapter 2 for more information.) Filter programs must be installed in */usr/lib/print*. It is recommended that any data filtering be performed directly by the driver. (Filters are programs that change the format of a data file; drivers communicate bidirectionally with the printer.) See chapter 12 for step-by-step instructions on adding new file conversion filters.

Step 5: Create Graphical Options Panel (Recommended)

This step is optional, but it is strongly recommended that you showcase the features of your printer by providing a graphical options panel program. (See Chapter 4, "Printer Graphical Options Panel," for details.)

Step 6: Package Software for Distribution (Required)

Package your Impressario software product for distribution. See Chapter 10 for detailed information.

Step 7: Verify Product on Server (Required)

Check that your product media will install the printer support files you have developed on an Impressario server. Run the Impressario test programs testipr and testiconfig to assist in verifying the installation. (See Chapter 9 for detailed information.)

Printing Application Programming Interfaces

To print a document on UNIX systems, you must submit a print job to one of the available spooling systems: the BSD spooling system (*lpr* command) or the System V spooling system (*lp* command). The System V spooling system is the default spooling system on all Silicon Graphics workstations. Figure 1-3 shows the general Impressario spooling architecture for the *lp* spooler. Note that only one of the two paths shown below (PostScript printer or raster printer) would apply. That is, the output is either to a PostScript printer or a raster printer, not both.
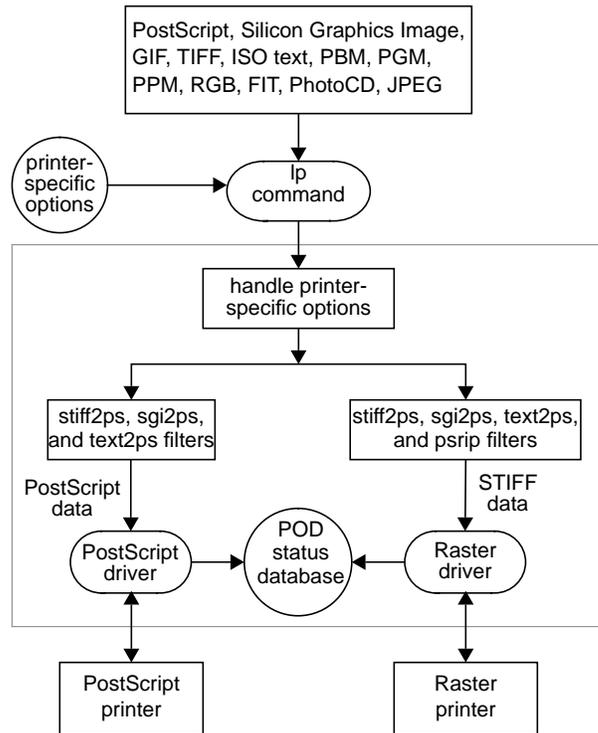
Figure 1-3   General Filter/Driver Architecture

Before Impressario, a true application program interface to the BSD™ and System V spooling systems was not available. Programmers had to create their own application programming interface or execute the *lp* or *lpr* command from their application.

9

Impressario provides the application program interfaces listed in Table 1-1 below.

Table 1-1   Printing Application Programming Interfaces

API	Interfaces to	Function
libspool	System V, BSD	Allows submission of a file or buffer to a printer
libprintui	System V only	Provides Motif *PrintBox* widget for printing from application
libpod	System V, BSD	Gets status information on the printers currently available

libspool API

In its simplest form, *libspool* allows you to submit a file or buffer to a printer. It also gives you control of spooling system options and printer-specific options.

libprintui API

The API *libprintui* is built on top of *libspool*. The *libprintui* library contains a widget, compatible with Motif, that you can incorporate directly into your application. If your Motif application needs printing capabilities, *libprintui* will provide you with all of the basic functionality for submitting a print job as well as access to setting and saving printer-specific options. As mentioned earlier, *libprintui* interfaces only with the System V spooling system.

libpod API

The *libpod* library is built on top of *libspool* and an ancillary system daemon, *podd*. The *libpod* library allows you to obtain detailed information about the capabilities of the printers currently available on the system. You can also get detailed status information about the printers. Most applications do not need to use *libpod*; however, for those that do, *libpod* provides a very powerful, network-transparent interface.

Figure 1-4 shows the relationship between an application program, the Impressario APIs, and the System V spooling system.
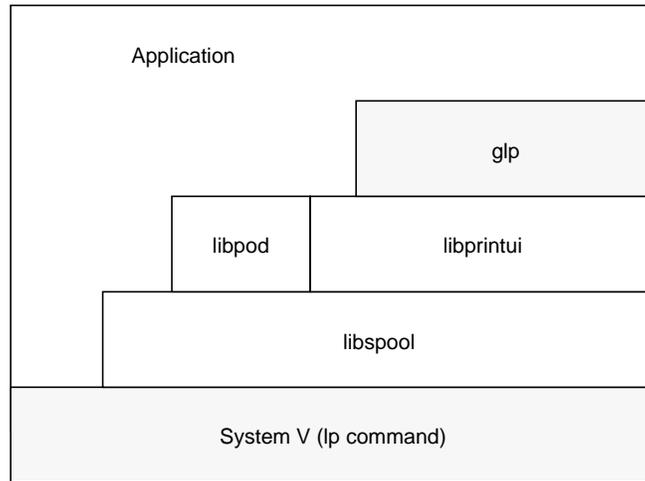
Figure 1-4   System V Spooling System Interface

Figure 1-5 shows the relationship between an application program, the Impressario *libspool* API, and the BSD spooling system.
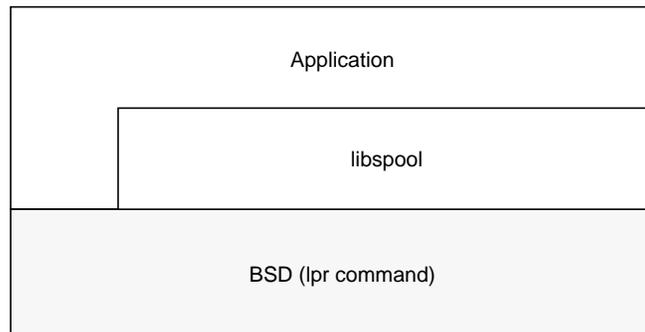
Figure 1-5   BSD Spooling System Interface

glp (PrintPanel)

The "print" subsystem available with the IRIX operating system contains the program *glp*, a graphical standalone print job submittal tool. While not a true API, *glp* can be used by your application to submit print jobs. It is a standalone wrapper around the *PrintBox* Motif widget in *libprintui*.

Printing Application Development

The following Impressario application printing solutions are available:

- Perform non-graphical print functions via *libspool*.

  Use *libspool*, the spooling system abstraction library, for all non-graphical interaction with the BSD or System V spooling system. The *libspool* library functions perform a large amount of work to ensure successful spooling system interaction, work that a developer may not wish to reproduce.

- Perform graphical print functions via *libprintui*.

  Use the *PrintBox* Motif widget provided by the *libprintui* library for graphical print job submittal. The widget provides a consistent job submission dialog across all applications and greatly reduces the amount of development effort required. If you use this library, you don't need to use *libspool* directly. *libprintui* encapsulates those calls.

- Obtain printer status via *libpod* (optional).

  Use *libpod* library calls to get printer status, configuration, and log information. Status queries must be made through *libpod* calls; attempts to obtain this information through other means have unpredictable results. In most cases, application developers should use the "standard" forms of the *libpod* functions (the "local" forms of the functions are for use by printer driver developers and are not network-transparent).

- Submit graphical print jobs via *glp*.

  If you are not using Motif, you may use *glp* to submit graphical print jobs from your application. If you use this method, you should have the *print* software package (shipped with IRIX 5.2 and later) as a prerequisite. While this method works, it is not recommended, as it is likely to be less efficient than using *libprintui*.

Complying With the Impressario Scanning Architecture

This section describes how to comply with the Impressario specifications for scanner driver developers and scanner application developers.

Figure 1-6 illustrates the run-time components of the Impressario scanner architecture. Note that scanner applications, scanner-specific options programs, and scanner drivers all link with *libscan.a*, which has separate modules for scanner applications and for scanner drivers.
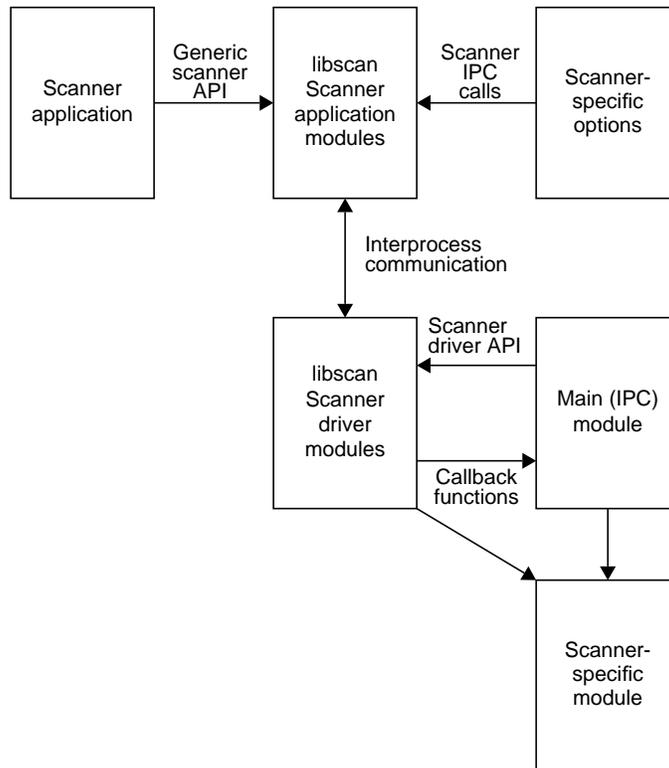
Figure 1-6   Scanner Run-Time Components

The main and scanner-specific modules of a scanner driver both register callback functions with *libscan*.

Figure 1-7 shows the interprocess communication of the Impressario scanner architecture.
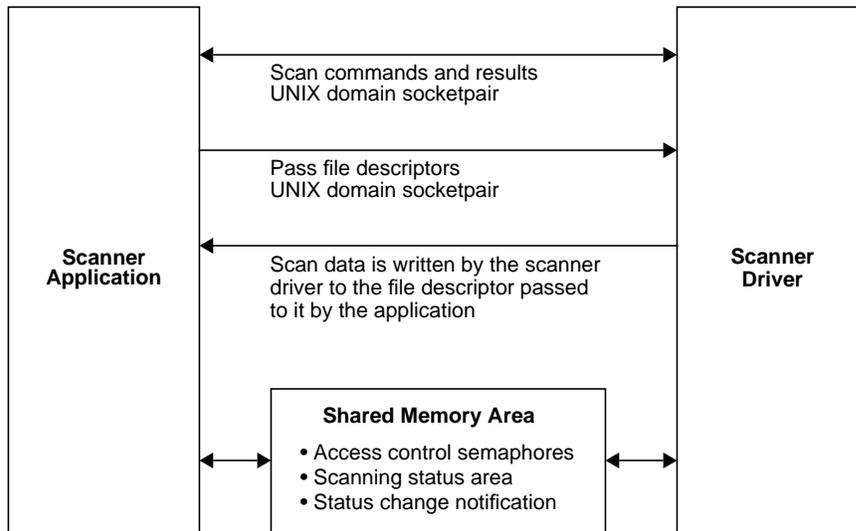
Figure 1-7   Interprocess Scanner Communication

The following sections describe how to achieve Impressario compliance for scanner drivers and applications.

Developing a Scanner Driver

Follow these steps to develop and integrate a scanner driver that complies with Impressario:

1. Develop the scanner driver. See Chapter 6, "Scanner Drivers," and Chapter 7, "Scanner-Specific Options," for detailed instructions.

2. Create a graphical options panel. This step is optional, but it is recommended that you showcase the features of your scanner through this mechanism. See Chapter 7, "Scanner-Specific Options," for detailed instructions.

3. Create distribution media that will install the driver on an Impressario server. Be sure to check that the media will install the scanner support files you have developed on an Impressario server. See Chapter 10 for detailed information.

Developing a Scanner Application

There are two ways for an application to interact with the scanning system:

1. Through *libscan*. This scanning system abstraction library provides a C-language API for scanners. Application programs can use *libscan* to retrieve data from any scanning device for which a driver exists. See the libscan(3) reference page for a detailed list of the *libscan* library functions.

2. As a *plug-in* module using gscan, which provides a complete graphical scanning interface. When passed the -**p** option, gscan writes scanned data to its standard output as a Stream TIFF file. After launching gscan with the -**p** option, applications can use the STIFF routines in *libstiff* to acquire the scanned data from the standard output of gscan.

Printer Drivers

This chapter provides an overview of printer driver processing, followed by a
detailed analysis and discussion of an example printer driver. The required
printer filter/driver options are also covered.

Printer Drivers

This chapter provides an overview of printer driver processing, followed by a detailed analysis of an example printer driver. The required printer filter/driver options and the reserved and unreserved printer filter/driver options are also covered.

The following topics are discussed in this chapter:

- "Printer Driver Processing" on page 21
- "Printer Driver Examples" on page 22
- "The Filter/Driver Specification and psrip" on page 25

Note: The PostScript interpreter, psrip, has been upgraded to an Adobe Level II Configurable PostScript Interpreter (CPSI). Adobe Systems, Inc. imposes strict limitations on the use of the Configurable PostScript Interpreter by developers. Please refer to Appendix H, "Using the Adobe Configurable PostScript Interpreter," for details.

Tip: Printer application developers can skip to Chapter 5.

Overview

A printer driver must perform the following tasks:

1. Receive and process an input file.

2. Send formatted data to the printer or to standard out. (The driver can use standard out when sending data to the generic drivers *phandler* and *nethandler.*)

3. Query the printer and receive status information. (This step is optional if *phandler* or *nethandler* is used.)

4. Read and write the printer object database (POD) files.

All printer drivers must read the POD files to determine printer-specific defaults and to maintain the active status file of the POD so that other clients can determine the printer's status. Chapter 5, "Printing Libraries," describes *libpod,* the library that provides an API to the POD. Appendix C, "Printer Object Database (POD) File Formats," defines the file formats for the printer configuration file, the printer status file, and the printer log file in the POD.

Printer Driver Processing

Figure 2-1 provides an overview of printer driver processing. Note that this is a simplified overview; the actual steps might be more complex.
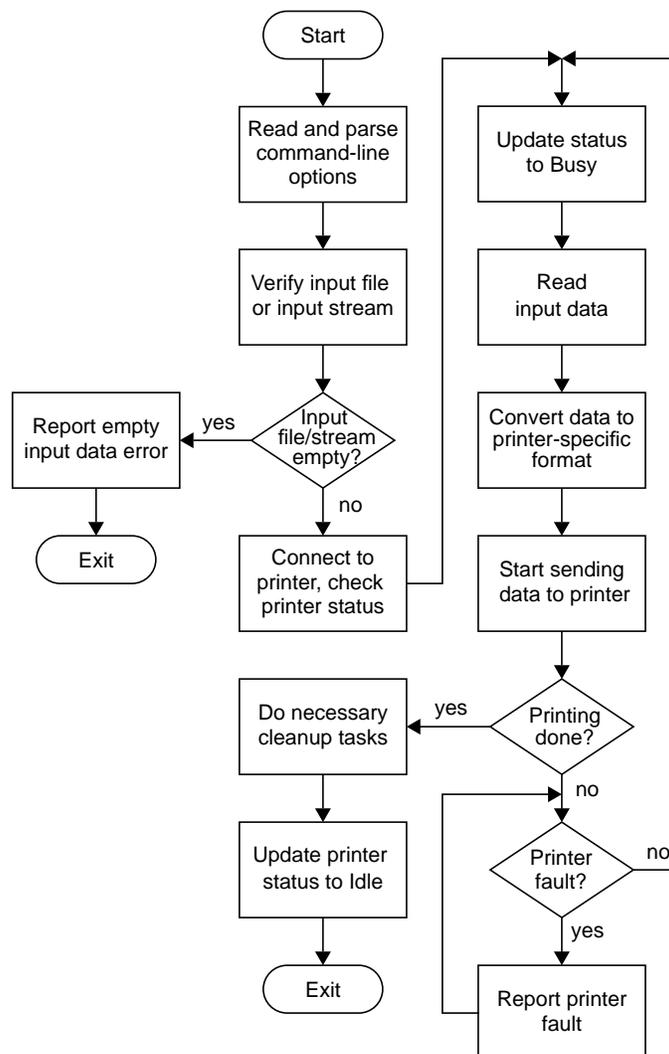
Figure 2-1   Printer Driver Processing Overview

Printer Driver Examples

Two example printer drivers are provided: *laserjetPJL* for HP LaserJet printers and *phandler* for generic parallel printers. The source code files for *laserjetPJL* are in the directory */usr/impressario/src/drivers/laserjetPJL* and the source code files for *phandler* are in the directory */usr/impressario/src/drivers/phandler*. Both drivers follow the steps shown in Figure 2-1, but the *laserjetPJL* driver is more complex because it also performs data compression. Because this added complexity is not necessary for the example, the focus is on the *phandler* driver in this chapter.

The example printer driver *phandler* checks the status of the parallel port while passing data through to the printer. It treats the input as a byte stream and updates the printer status file at various stages of processing.

Among the include files in this program is *pod.h,* which contains printer status error codes and other defined constants related to printing. The include file *plp.h* contains information related to the parallel port.

Program Invocation

The command-line interface for *phandler* is

```
phandler -P printer_name [options] [filename]
```

Arguments:

-**P** *printer_name* The name of the installed printer (required).

filename   The name of the file to send to the printer. Data is read from standard input if no filename is given.

options can be one or more of the following:

-**e**   Exit immediately on error. (The default is not to exit on error.)

-**w**   Suppress warning messages.

-**D**   Enable debugging. -**D** is the lowest level of detail, -**D** -**D** is the second level, and -**D** -**D** -**D** is the highest level of detail. The debugging information is appended to the log file.

-**L** *filename*   Use *filename* as the log file for debugging information and errors. The default is to use standard error, which the System V, Release 3 lp spooler redirects to the spooler log file.

-**R**   Reset the parallel port before sending data.

-**K**   Ignore interrupts.

-**I**   Enable sensing of the EOI pin. The default is to ignore the EOI pin.

-**B** *int*   Set the size of the internal transfer buffer to *int* bytes. The default is 1024. Larger buffers enable more page buffering, which may free upstream filters to do more processing while data is being sent to the printer.

-**s**   Invoke *phandler* to update the printer's status. Check the parallel port's status unless -**u** is specified. Update the paper size if -**S** is specified.

-**u**   Used only with -**s**. Don't check the parallel port's status. This allows the driver to ignore the parallel port while it is updating the POD status file.

-**S** *string*   Set the paper size (for example, A, A4, Legal) to *string* and update the POD status file for the printer.

-**d**   Send an ASCII Control-D (hexadecimal 0x4) at the end of the job.

To invoke *phandler* to update the POD data base, enter this command:

```
/usr/lib/print/phandler -P printername -s -u -K -S A
```

This use of *phandler* only updates the POD data base with the paper size. It ignores the parallel port status, and no information is sent to the parallel port. You normally use this command before invoking any filter that accesses the POD data base to insure that the filter gets the correct paper size.

After updating the POD data base with the paper size, you could send a file to the parallel port with *phandler* like this:

```
/usr/lib/print/text2ps -J printername textfile | \
/usr/lib/print/phandler -P printername -K -S A -d
```

Here, *text2ps* is used to convert an ASCII text file to a PostScript file. *text2ps* uses *printername* to access the POD and get printer-specific information, such as paper size. Other filters operate similarly to *text2ps* by accessing the POD for printer-specific information. This is why you run *phandler* to update the POD status before using a filter.

The output of *text2ps* is piped to *phandler*, which sends it to the parallel port while monitoring the printer's status and updating the POD.

Program Processing

This section contains a detailed discussion of the *phandler* driver. A more detailed example driver is supplied in the */usr/impressario/src/drivers/laserjetPJL* directory.

Note: The Impressario guidelines strictly require drivers to retry errors. Error exits are to be avoided whenever possible so that print jobs do not disappear before the user has a chance to fix the error condition.

Do the Initial Processing

1. Parse the command-line arguments with *getopts*.

2. Read the printer status file. If *phandler* cannot open the status file, it exits with an error message.

Open the Printer Port

1. Open the port. If there is an error and you requested exit on error when invoking *phandler*, then the program writes an error message and exits. Otherwise, when an error occurs, the program continues trying to open the port. It checks every *n* seconds (where *n* is the value of *error_retry_wait*) and writes an error message to the log file after each failed attempt to open the port.

2. If *phandler* successfully opens the port, it updates the current printer status to "Busy" in the printer active status file. If the command line contained the -**s** option (status only), then *phandler* exits after updating the status. (The kernel closes the port when it exits the program.)

Allocate and Set Up the Buffer

Allocate and set up the buffer. If *phandler* cannot allocate the buffer, the program exits with an error message. (It exits only because memory allocation errors are very rarely recoverable.)

Update libpod Status

Use the *fork* system call to create a child that updates the printer status.

Read, Process, and Send Data to the Printer

Begin reading from the input and begin sending data to the parallel port.

Your driver will probably need to process the input data into an appropriate printable format before sending the data to the printer.

1. If an unrecoverable error occurs while writing to the parallel port, write an error message to the log file, write the faulted state to the printer status configuration file, and exit.

2. If the input file is empty or unreadable, write an error message to the log file, update the status file, and exit.

The driver uses signal handling to ensure that it is not interrupted in "critical regions," during which an interrupt could destroy vital files or the printer state.

Cleaning Up and Exiting

When printing is complete, update the printer status configuration file to "Idle," terminate the child status process, and exit.

The Filter/Driver Specification and psrip

The command-line options listed in this section must be implemented as specified for any printing filter or driver to be compatible with Impressario. Switch letters have been chosen to maximize the intuitive correlation with function. Additional functionality beyond that listed here must use unreserved switch settings. Please note the following points:

•   All switches are case sensitive. That is, -**P** does not have the same meaning as -**p**.

•   Printer drivers must accept and ignore all reserved options that are not supported.

•   Printer drivers must conform to *getopts* conventions. (See getopts(2) for more information.)

- Multiple options on a single line have right-to-left precedence. For example:

  `-n 1 -n 2`

  has the same effect as

  `-n 2`

- The STIFF file passed to a printer driver may have command-line options embedded in it by psrip, a program that converts PostScript files to STIFF files. Drivers that accept input from psrip must accept and parse these command-line options. A good example is a PostScript file that contains a PostScript command to print 5 copies. psrip parses that PostScript command and embeds *-n5* in the STIFF file header. This header file information tells the printer driver to print 5 copies. The *laserjetPJL* example driver demonstrates how to parse command-line options embedded in a STIFF file. Appendix A, "Stream TIFF Data Format," details the STIFF header format.

Required Options

These switches must be supported by all drivers:

-**e**   Exit immediately on fault without waiting for faults to clear. The default behavior is to wait indefinitely for faults to clear, polling the device at the *error-retry* intervals specified in the POD configuration file. This option is used when only a quick query should be done.

-**s**   Update the status file. Exit only after successfully doing so. This switch has the highest precedence. If the -**e** switch is given, exit after one try at reading status.

-**w**   Do not report warnings in the status file. Report errors only.

-**D**   Enable debugging information. Optionally, more -**D** switches increase the level of debugging detail. For example, entering

          `-D -D`

          enables a second level of debugging detail. At least one level of debugging must be supported.

-**P** *string*   The value of *string* defines the printer name. The printer name is used to find the POD. The printer name is the name given to the printer at installation time. See the libpod(3) reference page for more information. This option is also a required option whenever the driver is invoked.

Impressario printer drivers must read the *libpod* printer object database to

- determine defaults
- maintain the active status portion of the POD database
- enable other clients to determine printer status

Reserved Options

The following options are reserved and are to be implemented by drivers whose hardware supports them, or by inline filters that process the options before the driver is invoked. You need not implement all options, but every driver must accept or ignore any unimplemented options on this list.

Raster-specific options include the following:

-f   Flip the image, as if in a mirror. The image is rotated horizontally about the y-axis. Useful for transparencies or decals.

-p *int*   Scale the image as if it were being printed on a device with the designated resolution specified in *int* pixels per inch. This is a convenience switch, since the same effect can be obtained by computing the appropriate scale factor for the image size and destination resolution.

-r *int*   Rotate the image counter-clockwise by an angle specified in *int* degrees. Values outside the range 0-359 should be accepted and modulo converted to a value between 0-359.

-z *float*   Zoom the image using proportional scaling, where the floating-point argument is nonnegative. Some values are given below:

        0.0   Do not zoom the image.

        0.5   Fill one-half of one page.

        1.0   Fill one page.

        Note: The image aspect ratio must be preserved. Future implementations may extend this to multiple pages. For example, *2.0* would mean fill a 2-by-2 page array.

Engine-specific options include the following:

-q *int*   Quality mode. Set the engine-specific quality mode. This should be a nonnegative integer, with greater values indicating higher quality.

-n *int*   The number of copies to be printed, a positive integer.

-t   Generate a test print. The test print should confirm that all marking media are present and functional.

-m *int*   Manual feed request. Wait *MediaWaitTimeout* seconds for manual feed. Give up after *MediaWaitTimeout* seconds and print anyway on the available media. See the POD for these values. Giving up is important for shared printers.

-o *int*   Request a specific output medium:

        0   paper or a reflective medium

        1   transparencies

        Other media types may be supported; see the libpod(4) reference pages.

Output-specific options include the following:

-L *filename*   Log errors to *filename* instead of standard error. The file specified should be opened in append mode. If the file cannot be opened, errors should be reported to standard error instead.

-O *filename*   Output data to *filename* instead of the device port or standard output. The file specified should be opened. If the file cannot be opened, data should instead be written to the device or standard output, as appropriate. If this option is used, all status reporting is disabled, because the printer driver is not communicating with the actual device.

Unreserved Options

The switches listed below are not reserved and can be used for device-specific options:

- Lowercase: a, b, c, d, g, h, i, j, k, l, u, v, x, y
- Uppercase: A, B, C, E, F, G, H, I, J, K, M, N, Q, R, S, T, U, V, W, X, Y, Z

Printer Model Files

This chapter discusses the printer model files and describes the modifications that must be made by printer vendors to the printer model file template.

Printer Model Files

This chapter discusses the printer model files and describes the modifications to be made by printer-driver developers to the printer model file template.

The following major topics are discussed in this chapter:

- "Overview" on page 32
- "Command-Line Arguments" on page 32
- "Template Model File Execution" on page 33
- "Printer-Specific Options" on page 37
- "Developer-Supplied Model File Additions" on page 39

Overview

Model files are Bourne shell scripts that form an interface between the System V, Release 3 spooling system and the printer. Each printer has its own model file, which is customized by the printer installation tools when the printer is installed. The customized copy of the printer model file is the *interface file.* The interface file is invoked by the spooling scheduler *lpsched* when the printer is ready to accept a new print job. The interface file sets up printer-specific options, calls filters, and invokes the printer driver.

System V model files differ greatly from BSD */etc/printcap* entries. The reader unfamiliar with System V spooling should refer to the *IRIX Admin: Peripheral Devices* manual for more information.

Command-Line Arguments

The model file expects these command-line arguments from *lpsched* in the order listed:

1	job sequence ID number
2	user login name
3	job title
4	number of copies to be printed
5	-**o** printer options
6-n	name(s) of the file(s) to be printed

The end user does not invoke the model file manually; it is invoked only through *lpsched.* If you want to check for gross syntax errors, you can do this quickly by running the model file with dummy arguments such as:

```
laserjetPJL_model a b c d e
```

Not all errors can be found this way, since the model file does not run to completion, but this does provide a quick test for gross syntax errors. If errors are found, putting the -**x** flag at the end of the first line in the model file may help debug the error. See the sh(1) reference pages for more details on debugging Bourne shell scripts.

Template Model File Execution

This section explains how the Impressario model files work.

The source code for the template *laserjetPJL_model* file is located in the directory */usr/impressario/src/models.* The main steps in the template model file are shown below and described in more detail in the following sections:

1. Declare global and external variables and define convenience functions.

2. Interpret and store the command-line options.

3. Verify that the prerequisites are in place.

4. Set the output device driver, which can be either a driver for the parallel port or a driver for a network card that is installed in a printer.

5. Invoke the driver to update the POD status file with the current printer state and the current job setting.

    Note: This is an important step. All filters use the status file to determine what parameters to use. The driver must ensure that the POD status file is absolutely up to date before the first filter is run, or a change in printer status between jobs could compromise the next job.

6. Start the active icon tagging subprocess.

7. If pages stack face down, print a banner page if requested.

8. Use filters to convert the submitted file to the data type required by the printer driver.

9. Invoke the printer driver with the converted data.

10. Repeat steps 8 and 9 for each file to be printed and report any errors encountered to standard error.

11. If pages stack face up, print a banner page if requested.

12. Clean up after the job and exit with an appropriate exit code.

Declaring Variables

The steps given below tell you what variables to declare. Note that variable names are in all uppercase letters. Those variables are exported for use by the filters invoked when converting input files into printable data. See the fileconvert(1) reference page for more information.

1. Define *NAME* and *TYPE*. If the type is not defined, then the model file writes an error message and exits.

2. Append the standard error output from the filters and driver to the spooler log file.

3. Define the file paths, directory paths, and Boolean flags used in the model file.

4. Define the locations and options for all filters. (The model file checks to make sure that the printer-specific driver exists and that the spooler log file is writable. If these conditions are not met, the model file writes an error message and exits.)

Defining Convenience Functions

The next portion of the model file contains various convenience functions used within the model file. The convenience functions listed in Table 3-1 are routines contained within the model file. These routines are used for parsing the -**o** options and for various utility functions.

Table 3-1   Convenience Functions

Function Name	Description
BeginTagging()	Sets up the tagging job that monitors the printer's status.
CleanUpAfterJob()	Does any cleanup needed at the end of the job.
EndTagging()	Ends the tagging of the printer.
ParseOptions()	Parses the -**o** command-line options and sets appropriate variables. Expects the command-line options string as the first argument.
PrintBannerPage()	Prints a banner page.
PrintMessage()	Prints its arguments as text on the printer. Used to report errors to the user. Accepts any number of arguments.

Table 3-1 (continued)   Convenience Functions

Function Name	Description
ReportBadFile()	Reports unknown file type to the printer and to the spooler log. Expects the filename as the first argument and the file type as the second argument.
ReportUnknownOption()	Reports an error when an unknown option is parsed. Expects the unknown option as the first argument.
SetCancelTrap()	Sets up the trap command for the signals SIGHUP, SIGINT, and SIGTERM.
SetDebug()	Turns on debugging mode for all filters and drivers.
SetDeviceDriver()	Sets the output device driver to a parallel port driver or to a driver for a network card installed in a printer.
TestExitStatus()	Interprets exit status from the last command and sends error message to both the spooler log and the printer if needed. Expects the exit code of the last command as the first argument, a string describing the last command as the second argument, and the file in question as the third.

Processing Command-Line Arguments

The template model file follows these steps to process command-line arguments:

1.   Retrieve the command-line arguments.

2.   Set up the cancellation signal handler for SIGHUP, SIGINT, and SIGTERM. This allows the user to cater the job cancellation signal sent by the spooler and clean up.

3.   Parse the options passed with the -**o** switch.

4.   Set the device driver.

5.   If the *verbose* switch is set, send a message to the spooler log that the job has begun filtering.

6.   Call the printer status and communications driver to update the status database before job filtering begins.

7.   Start the tagging subprocess to tag the printer's workspace icon with the appropriate print-engine type and status.

Printing Banner Page

The model file prints a banner page is printed if the *banner* switch is set. (The banner page may be printed last for face-up stacking printers.)

Using Filters to Process Files

1. The *fileconvert* command automatically determines the file type of each file using the Indigo Magic file typing rule database. (See the reference pages for ftr(1) and *fileconvert*.) If a fast text path exists, and the user has not requested options that require the slower path, then the file is converted using the fast text path. Otherwise, the normal *fileconvert* path is used. See the template model file for specifics.

2. *fileconvert* produces a shell command string that, when executed, produces the requested data type on standard output.

3. If the file could not be converted, **ReportBadFile()** is called; otherwise the *fileconvert* shell program is executed and the output piped to the printer driver.

4. **TestExitStatus()** is called to test whether the driver reported an error.

Cleaning Up and Exiting

1. After the printing is complete, perform any needed cleanup.

2. End the tagging subprocess.

3. Append an ending message to the log file if the *debug* switch is set.

4. If there were unsupported file types in the list of files to be processed, exit the model file with an easily understood error code.

Printer-Specific Options

The -**o** option to the spooler allows model files to accept a variety of non-spooler options. These are the options specified with -**o** on the *lp* command line. It is these options that are produced by the graphical printer options panel.

Impressario defines a number of general file filtering options for the convenience of the end user, and reserves those option names.

The reserved options are listed in Table 3-2 in alphabetical order. Their meanings should not be changed by printer driver developers. Developers should choose short, understandable option names for any additional options. These options may be seen by end users, so they should not be verbose.

Table 3-2   Reserved Option Names

Option Name	Description
banner	Prints a banner page.
bestfit	Uses the best fit orientation for image files (default is *on*).
bottommargin	Sets the bottom margin size for text.
columns	Sets the number of columns on the page for text.
debug	Causes filters to report debugging information (default is *off*).
duplex	Turns duplex on or off.
flip	Flips the image, producing a mirror image.
fontname	Uses a specified font name for text.
fontsize	Uses a specified font size for text.
gamma	Uses a specified value for image gamma correction.
gaudy	Uses a fancy page header for formatted text.
halftone	Sets the halftone or dithering option to be used.
intray	Specifies the input media source (same as the tray option).
landscape	Uses landscape page orientation for text.
leftmargin	Sets the left margin size for text.

Table 3-2 (continued)   Reserved Option Names

Option Name	Description
manpage	Indicates the preformatted manual page to be printed.
manual	Sets manual feed.
nobanner	Sets the "do not print a banner page" option.
nogaudy	Sets the "do not use fancy page header for formatted text" option.
noverbose	Sets the "do not print verbose messages in the spooler log file" option (same as the -h option).
numberpages	Sets the number of text pages.
outtray	Specifies the output media source.
papersize	Selects the paper size.
portrait	Uses portrait page orientation for text.
ppi	Scales the final image size to match the specified original image resolution, in pixels per inch.
psevenpage	Prints even pages only (PostScript).
psoddpage	Prints odd pages only (PostScript).
pspagerange	Prints the specified page range (PostScript).
psreversepage	Reverses the page order (PostScript).
resolution	Sets x and y resolutions (for resolution-switchable printers).
reversepages	Reverses text document page order (prints the last page first).
rightmargin	Sets the right margin size for text.
rotate	Rotates the image clockwise in 90 degree increments.
topmargin	Sets the top margin size for text.
tray	Specifies the input media source.
verbose	Records debugging messages in the spooler log file.
zoom	Scales the image to fit the page (default is a scale of 1.0).

Developer-Supplied Model File Additions

Printer-driver developers must customize the model file template supplied by Silicon Graphics to the specifications of their printers. The template model is located in the file */usr/impressario/src/models/template_model.* This code is the same as the code in the *laserjetPJL_model* file so that developers can have a working, debugged example from which to begin.

The following ten items must be specified by you. They are listed and explained in the order in which they appear in the model file template.

1. Printer name.

2. Device interface.

3. Printer type.

4. GUI class.

5. Printer-specific filter/driver.

6. Debug routine.

7. Cleanup routine.

8. Printer-specific banner page.

9. Printer-specific filtering options.

10. Fast path for text.

Note: All items that must be modified by the developer have been marked with "#XXX" in the model file. Search for and remove these markers as you progress to be sure all necessary modifications have been made.

Printer Name

The *NAME* variable identifies the real printer name, such as "HP DeskJet 500C." *NAME* contains the string that is presented to the end user in the Printer Manager's graphical printer install tool. Multiple *NAME* variables are allowed on separate lines if the model file can support more than one printer.

39

Device Interface

The *DEVICE* variable identifies the hardware interface where the printer is attached. The value is used by the printer install tool to present models by connection type. Multiple connections are supported. (Use multiple lines for multiple devices; that is, simply repeat the line for each different device.) The values currently allowed are SERIAL, SCSI, CENTRONICS, NETPRINTER, and REMOTE. Obsolete types that should not be used are VERSATEC and TEK.

Printer Type

The *TYPE* variable identifies the printer type. This information is used by the Print Manager, routeprint, *libspool* functions, and other system software, including the active icons subsystem. Table 3-3 shows the values allowed for the *TYPE* variable. Obsolete types are Dumb, Color, and PostScript.

Table 3-3 Printer Type Specifications

TYPE Value	Data Types Accepted	Printer Examples
Raster	At least text, .SGI image, PostScript	HP LaserJet
ColorRaster	At least text, .SGI image, PostScript	Tektronix Phaser II SX
Plotter	HP-GL only	HP 7550A
ColorPostScript	At least text, .SGI image, PostScript	Tektronix Phaser II PXi
MonoPostScript	At least text, .SGI image, PostScript	LaserWriter II NTX

GUI Class

The value of this variable is the name of the resource file for the graphical options panel. This value must match the **#define** GUI_CLASS in the options panel *gui_class.h*. If an options panel is not provided, do not specify a value for this variable.

Printer-Specific Filter/Driver

The file *laserjetPJL.c* contains an example of a complete printer driver. This driver is used in the */usr/impressario/src/models/laserjetPJL_model* file. For comments about the model file, see the */usr/impressario/src/models/template_model.README* file.

Debug Routine

To turn on debugging for developer-supplied filters, modify **SetDebug()** to set the debug switch (-**D**) for each filter.

Cleanup Routine

Some developers may need to perform some cleanup at the end of each job. If so, add this cleanup to the routine **CleanUpAfterJob()** in the model file. To make this step easier, the use of temporary or intermediate files is strongly discouraged.

Printer-Specific Banner Page

It is recommended that you do not modify this routine. However, to create a customized banner page that uses printer features such as page counts or graphics, modify **PrintBannerPage()** as required. The banner page should not unduly slow down the printing process, and customized banner pages should include at least as much job-specific information as the default banner page: job ID, user name, job title, date and time, and the filename(s).

If your printer is unlikely to be shared, or has high per-page costs, you may want to turn off banner pages by default. In this case, set the variable *banner* to zero (0). However, users can still choose to print a banner page.

Printers that stack pages face down should print the banner page before any files, while those that stack pages face up should print the banner page last. This is handled automatically if the *faceup* variable is set appropriately.

Printer-Specific Filtering Options

The **ParseOptions()** routine parses the **-o** command-line options and sets appropriate global variables. This routine contains many general options, and developer-specific options can be added if required. However, developer-supplied options must not duplicate Silicon Graphics reserved words. See "Printer-Specific Options" on page 37 for a complete, alphabetized list of existing options.

It is recommended that any developer-specific options be full-word names to improve the readability of the stored settings files and to reduce name-space conflicts. This also aids users who use command-line interfaces to printing. However, option names should be kept brief.

Fast Path for Text

You should modify the path in the file filtering section of the model file to use native text support if your printer supports native text printing that is faster than PostScript printing. Users, however, must still have access to all PostScript formatting options. Do not disable the slower PostScript path.

Printer Graphical Options Panel

This chapter explains how to implement a graphical options panel that visually showcases your printer's features and improves its usability.

Printer Graphical Options Panel

This chapter discusses the graphical options panel, which visually showcases a printer's features.

The following major topics are discussed in this chapter:

Overview

Impressario provides the PrintBox widget for submitting print jobs from Motif applications. This widget is contained in the library libprintui(3X). The PrintBox widget is used by the *glp* or *PrintPanel* command and a number of other applications to provide their printing capability. In addition to providing graphical selection of System V print job submission options, the widget provides the *graphical options panel* for graphical selection of printer-specific options.

A graphical options panel allows the printer-driver developer to showcase the unique features of a printer in an intuitive graphical panel. The graphical options panel program is invoked by the PrintBox widget in applications, by the end user via *glp*, or by the Printer Manager in the System Toolchest. The *Graphical Options Panel Specification* located in the */usr/impressario/doc* directory provides the information necessary to create and integrate a graphical options panel. Graphical options panels are standalone executable programs that are stored in a standard directory known to the Impressario printing tools. Therefore, the graphical options panels are automatically available to users of Impressario printing tools. The rules for developing a graphical options panel are straightforward and do not require any interprocess communication or similar complex procedures. In fact, we strongly discourage any network dependencies because not every Impressario printer is on a network.

Graphical Options Panel Layout

A graphical options panel most often consists of a single window. This window usually contains two sections, an options section and an action area. We strongly recommend that you keep options panels simple to avoid both complex code and complicated documentation.

The options section contains all printer-specific option controls. This area is often divided into groups of option controls, where each group represents a specific input file type. Most options sections contain controls for text files, bitmap image files, and PostScript files. The options section also contains a general options section. Because there are often a large number of controls in the options section, you should use a scrolled window to limit the graphical options panel window to a reasonable size. Figure 4-1 shows an example of a graphical printer options panel.
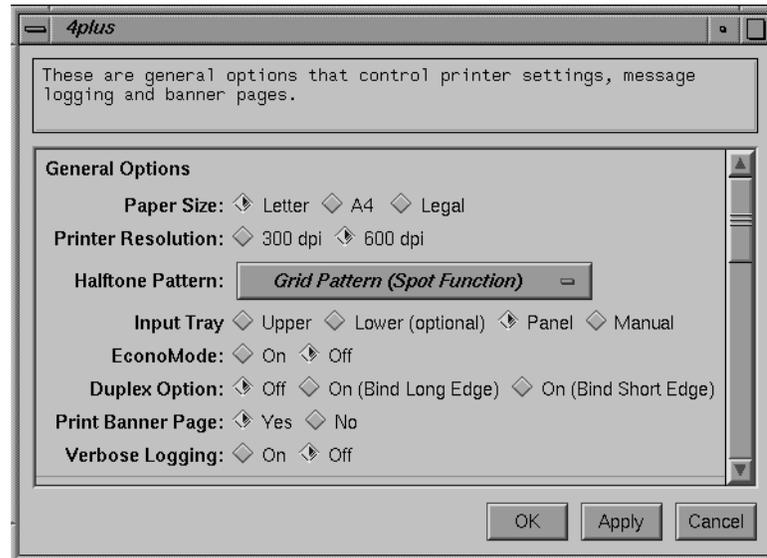
Figure 4-1   Graphical Printer Options Panel

The action area is located at the bottom of the graphical options panel window and consists of a number of push buttons. The first three are required:

OK   Output the option string to standard out and terminate the program.

Apply   Output the option string to standard out; do not terminate the program.

Cancel   Terminate the program without any option string output.

Help   Provide printer-specific options help.

Group buttons together on the right side of the action area. The width of all buttons should be equal. The leftmost button should be *OK* followed by *Apply*, *Cancel*, and *Help*. Place any additional buttons between the *Apply* and *Cancel* buttons. Note that the supplied template makes this happen automatically.

Options Handling

If printer-specific options have been passed to the graphical options panel on the command line, the program must interpret these options and initialize its options-section controls to reflect the command-line options. Options not recognized by the graphical options panel must be preserved and prepended to the output option string.

When the *OK* or *Apply* button is activated, the graphical options panel program must form a valid System V printer option string based on its GUI settings, and print this string to its standard output.

Graphical Options Panel Development

The directory */usr/impressario/src/gui_models* contains example source code for a graphical options panel. Begin with this code and add your printer-specific options. Do not start from scratch! You will waste valuable development time and may create inconsistencies. Benefit from others' experience and begin with the template.

Create the graphical options panel with the OSF/Motif UI™ toolkit. Starting with Impressario 1.2, your graphical model file can have its own application resource file. While the resource file can be given any name, Silicon Graphics recommends that the name represent the printer model name and have its first letter capitalized. For example, the application resource file for the HP LaserJet graphical model file is called *LaserJetPJL*. The name chosen for the resource file must be specified in the *gui_class.h* header file and must also be specified in the printer model file as the value of the *GUI_CLASS* variable. Your installation media must place the resource file in the directory */usr/lib/X11/app-defaults*.

To match the look and feel of the *PrintPanel* program (*glp*) and other graphical options panels, the application resource file should include the following resources:

*useSchemes: all

*schemeFileList: SgiSpec

*sgiMode: True

To facilitate localization of the options panel for international customers, all label strings and messages should be placed in the application resource file rather than being hard-coded into the program.

The graphical model files must consist of a single executable program and its application resource file. The model files are restricted to these because the printer installation tools install only an executable and its resource file during network printer installation.

Graphical Options Panel Naming

The graphical options panel must be given the exact same name as its printer's model file, followed by the suffix *.gui*. For example, if the printer model file for an HP DeskJet 500C printer is called *deskjet_model*, then the graphical options panel must be given the name *deskjet_model.gui*. If the graphical options panel is given a name that differs from the model filename, it will not be installed by the printer install tools when a new printer is installed on a system.

Graphical Options Panel Installation

All graphical options panel programs must be installed by your installation media in the directory */var/spool/lp/gui_model* or */var/spool/lp/gui_model/ELF*[1]. These directories are where the printer installation software searches for the programs. COFF executables must be installed in the directory */var/spool/lp/gui_model*. The executable should be owned by *lp* and should be a member of the group *lp*. The file permissions of the executable should be set to *0755*.

Note: You must always supply an ELF executable. COFF executables are not supported in IRIX 6.2, and are necessary only if you wish to support IRIX 4.0.5 systems.

––––––––––––––––––––––––––––––––

[1] All programs compiled on systems running IRIX 5.0 or later are ELF executables.

Invocation by the PrintBox Widget

The graphical options panel is invoked by the PrintBox widget or the Printer Manager. The graphical options panel is always invoked with the command-line arguments listed in Table 4-1. Note that the options are almost identical to those passed to your spooler model file.

Table 4-1   Command-Line Arguments

Argument	Description
argv[0]	Printer name
argv[1]	User name
argv[2]	Filename(s) string
argv[3]	Printer-specific option string (optional)
argv[4-*n*]	Xt options (optional)

Any **Xt** options may be specified as *argv[4]* or greater. If there are no printer-specific options for *argv[3]*, the empty string ("") is passed as *argv[3]*. If no filenames are specified, the empty string is also passed as *argv[2]*.

Standalone Invocation for Testing

To test a graphical options panel during development, it may be run as a standalone executable from the UNIX command line. Invoke the graphical options panel by entering

```
[executable name] [any valid user name] ""
```

For example, if the executable program is called laser*jetPJL_model.gui*, and "*joe*" is a valid user name on the system, then the program can be executed from the command line by entering:

```
laserjetPJL_model.gui joe ""
```

When debugging, it often helps to invoke the graphical options panel from the shell and repeatedly test the options string that is generated when the *Apply* button is pushed. Once the options string is properly generated, test for proper parsing of the input options by invoking the graphical options panel with various options strings, especially those output by your panel. Be especially careful that all widgets are set properly when, upon startup, the panel is passed a string that sets options to non-default values.

Termination by the PrintBox Widget

Applications terminate the graphical options panel using a SIGHUP signal. The graphical options panel should exit promptly upon receipt of this signal.

Additional Information

For additional information, please read the online specification found in */usr/impressario/doc/gui_model.spec*.

The template was intended to make the creation of these panels easier. If you find the process difficult, please check the template documentation and the convenience routines in the support files. There may be a solution already provided.

Printing Libraries

This chapter describes the printing libraries used by Impressario printer drivers, filters, and applications.

Printing Libraries

This chapter describes the printing libraries used by Impressario printer drivers, filters, and applications.

Three printing libraries are described in this chapter:

- "The libspool Library" on page 56 is a C application program interface (API) to the UNIX printer spooling systems.

- "The libprintui Library" on page 58 is a C-language API to the PrintBox, a widget compatible with Motif.

- "The libpod Library" on page 64 is a C-language API to the printer object database (POD).

In addition to the above libraries, there are two libraries described in the appendices that are also used by printer drivers and filters:

- Appendix A, "Stream TIFF Data Format," describes *libstiff*, a C-language API for reading and writing the STIFF (Stream TIFF) data file format.

- Appendix B, "Silicon Graphics Image File Format API," describes *libimp*, a C-language API for reading and writing Silicon Graphics Image format files.

The libspool Library

The *libspool* library is a C application program interface to the UNIX printer spooling systems. There are two common UNIX printer spooling systems, System V and Berkeley Software Distribution (BSD). While these spooling systems provide essentially the same capabilities, each has its own command set and neither provides a C-language API. The *libspool* library provides a single, common API to both spooling systems. The functions provided by *libspool* include submission and cancellation of print jobs, and control and reading of print queues.

Compiling Programs With libspool

Programs that call *libspool* functions must include *spool.h*, the header file in the */usr/include* directory. Use the following **#include** directive:

```
#include <spool.h>
```

The programs must also link with the *libspool.a* library located in */usr/lib*. Here is an example of the complete *cc* compiler command line:

```
cc -o myprog myprog.c -lspool
```

libspool Library Functions

Table 5-1 lists the *libspool* functions by purpose.

Table 5-1   Summary of libspool Functions

Task	Function Name	Purpose
Spooling System Selection	SLSetSpooler()	Set the default spooling system.
	SLGetSpooler()	Get the default spooling system and available systems.
Printer Information	SLGetPrinterList()	Get the list of registered printers
	SLGetPrinterInfo()	Return information about a printer.
	SLGetDefPrinterName()	Get the name of the default printer.
	SLGetPrinterSettings()	Get spooler and printer settings.
Option Management	SLSysVGetSpoolerOptions()	Get System V spooler options.
	SLSysVGetPrinterOptions()	Get System V printer options.
	SLSysVSaveSpoolerOptions()	Save System V spooler options.
	SLSysVSavePrinterOptions()	Save System V printer options.
Print Job Submission	SLSubmitJob()	Submit a job for printing.
	SLSubmitJobFd()	Submit the contents of the file specified by a file descriptor.
	SLSubmitJobBuf()	Submit contents of specified buffer.
	SLSubmitJobSimple()	Submit a print job using default values for all printing options.
Print Job Cancellation	SLCancelJob()	Cancel a queued printer job.
Printer Queue Information	SLGetQueue()	Report the printer queue contents.
Printer Queue Control	SLSetSpoolerState()	Set the spooling system printing and queueing state.
	SLGetSpoolerState()	Get the spooling system printing and queueing state.
Execution Error Handling	SLPerror()	Print a libspool execution error message to standard error.
	SLErrorString()	Get a libspool execution error msg.
	SLGetSpoolerError()	Get spooling system error info.

The libprintui Library

The *libprintui* library implements a graphical user interface (GUI) for printing. The library provides PrintBox, a complete solution for print-job submission, to application developers. A widget that is compatible with Motif, PrintBox eliminates the need to create custom printing solutions for each application. This widget saves application developers time and effort, and makes it easier for them to provide a robust, complete printing interface.

Using the PrintBox widget in an application benefits the end user in several ways. First, PrintBox provides a consistent interface to the printer spooling system across the varied applications on Silicon Graphics systems. Second, users can set print job options, such as number of copies, through a graphical interface, rather than through obscure command-line option flags. Finally, PrintBox uses the printer graphical options panel to provide a mechanism for the setting and saving of printer-specific options.

The PrintBox widget can be used in a number of different configurations and can accept a child manager widget to allow the display of application-specific options. The widget provides built-in System V print job submission via the *libspool* library. (See "The libspool Library" on page 56 for more information.) The developer can also perform application-specific processing before a job is submitted to the printing system. A variety of callback lists provide user and spooling system feedback. A print job can be submitted as a filename, as a file descriptor, or as a pointer to the buffer. The default form of PrintBox includes the following items:

- A print file entry text field (for file-based jobs)

- A scrolling list of available printers

- Print option controls

- The following action-area push buttons:

Print	Submits the specified file or buffer for printing by the spooling system.
More Options …	Accesses the graphical options panel for the currently selected printer.
Save Options …	Saves printer and spooling system options.
Cancel	Normally used to pop down the PrintBox widget when the widget is used as a pop-up dialog.
Help	Calls the functions on the *helpCallback* list.

There are also four unmanaged buttons, *User1* through *User4,* positioned between the *Print* and *More Options…* buttons These buttons, invisible by default, become visible when explicitly managed by your application. The PrintBox widget also accepts one child process as a work area. This area can be used for application-specific printing controls such as page range.

Example Widget Configurations

Figure 5-1 through Figure 5-4 illustrate four widget configurations:

- the default configuration
- without a filename entry box
- without an options box
- with a child process
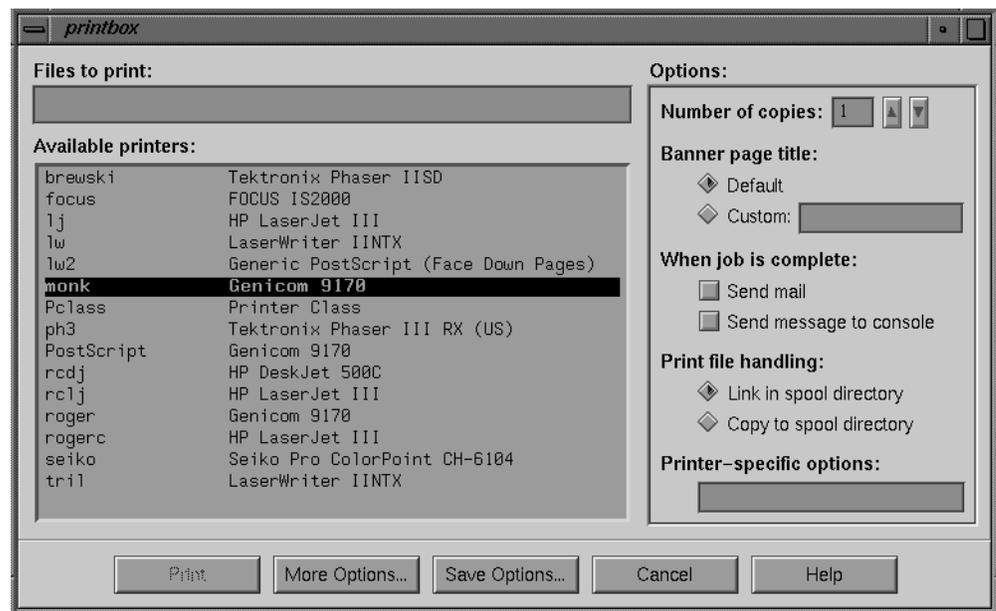
Figure 5-1   PrintBox Widget: Default Configuration
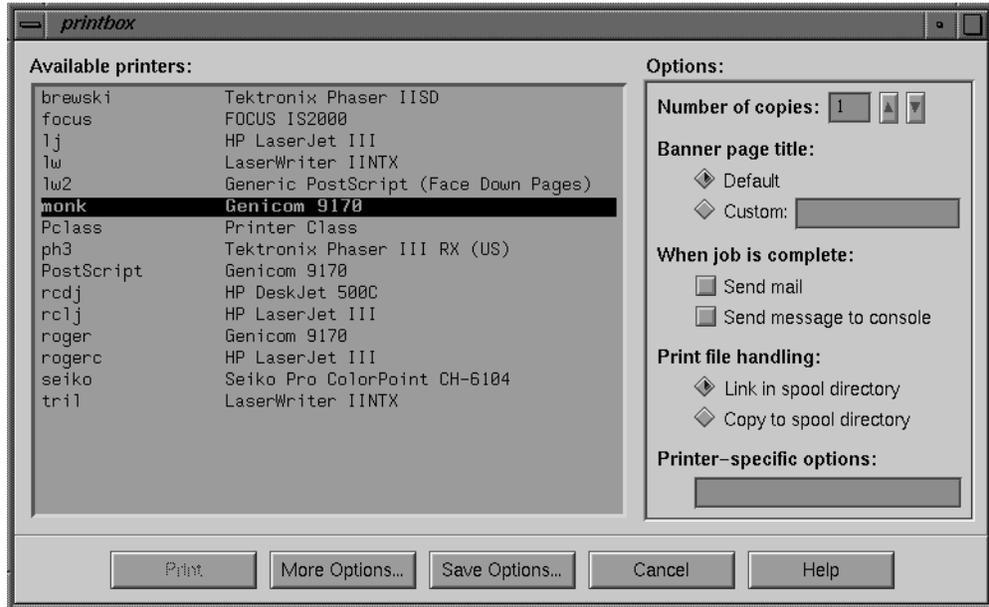
Figure 5-2   PrintBox Widget: No Filename Entry Box
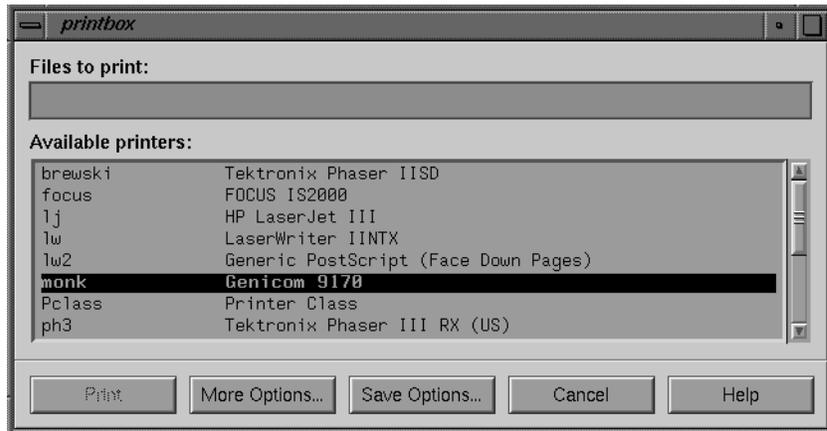
Figure 5-3   PrintBox Widget: No Options Box
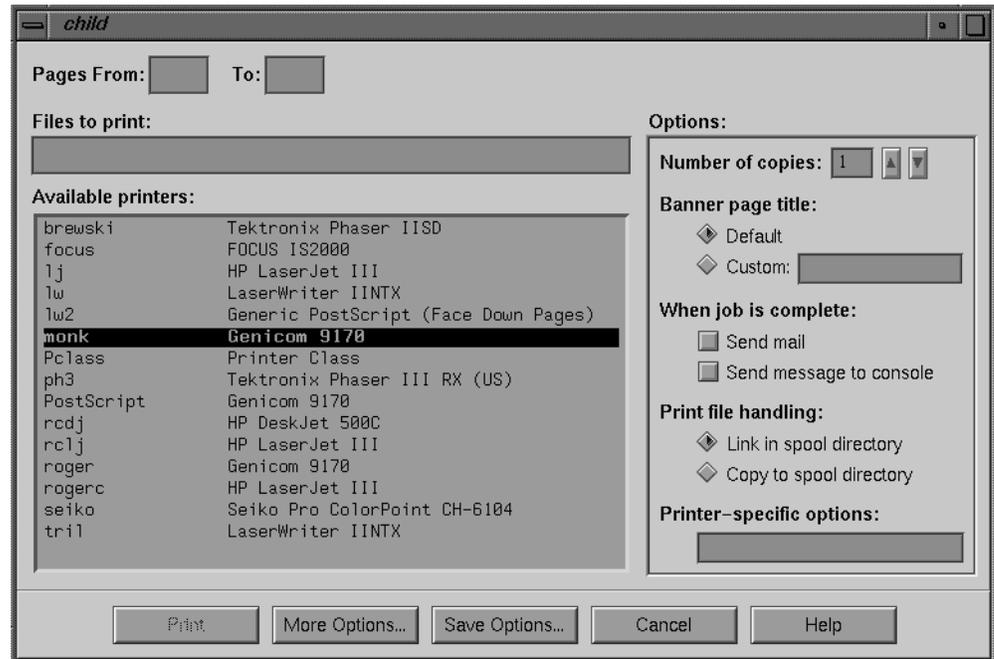
Figure 5-4   PrintBox Widget: With a Child Process

Compiling Programs With libprintui

Programs that call the *libprintui* functions must include the header file
/usr/include/Sgm/PrintBox.h and must link with the following libraries in the order shown:

```
... -lprintui -lspool -lXm -lXt -lXll -lgen ...
```

The link order is important for proper link-time name resolution.

Note: Programs that subclass from the PrintBox widget must also include
/usr/include/Sgm/PrintBoxP.h.

Library Functions Listed by Purpose

The *libprintui* functions are listed in Table 5-2. The PuiPrintBox(3X) reference pages provide detailed information on the PrintBox widget.

Table 5-2   Summary of libprintui Functions

Task	Function	Purpose
Widget Instantiation	PuiCreatePrintBox()	Create a PrintBox widget.
	PuiCreatePrintDialog()	Create a PrintBox dialog.
Widget Component Access	PuiPrintBoxGetChild()	Access a PrintBox widget component.
Widget Action Functions	PuiPrintBoxDoPrint()	Invoke PrintBox printing.

Example Program

The example program, printbox, instantiates a simple PrintBox widget. The directory */usr/impressario/src/examples/libprintui* contains the source code for this program, while the directory */usr/impressario/bin/examples/libprintui* contains the executable version.

To invoke the example program, enter:

```
printbox
```

Initial Program Processing

The printbox program begins by setting the program instance name and initializing an X Window System™ connection. Next, the program creates the PrintBox widget with a call to the *libprintui* library function **PuiCreatePrintBox()** and adds the widget to the parent's managed set.

Add Callbacks

The program now adds the following callbacks:

- the *Cancel* button exit routine

- a help dialog display

- a routine to print job information and the job ID to standard output and terminate the program

- a display of error messages from the PrintBox widget (this uses the function **SLGetSpoolerError()** from the *libspool* library)

Realize All Widgets

A call to **XtRealizeWidget()** now realizes (creates a window for) the parent widget, created by the earlier call to **PuiCreatePrintBox(),** and all child widgets.

Process Events

The program now begins an event loop. It obtains the next event from the X event queue and dispatches the event. If an early error has occurred, the error is handled and the loop continues until the application exits.

Additional Examples

Refer to the directory */usr/impressario/src/examples/libprintui* for additional sample program source code.

The libpod Library

The *libpod* library provides printer driver developers with an API to create and maintain a printer object database (POD) and provides application developers with the means to acquire detailed information about a printer, even across the network. A POD contains information on the current printer configuration, status, and job history of a single printer. Each printer driver installed on a system maintains its own POD on that system. All interaction with a printer's POD must be done through the *libpod* API. Do not modify the POD files directly. To create an initial set of POD files, refer to Appendix C, "Printer Object Database (POD) File Formats," and the examples provided in */usr/impressario/src/data.*

POD Files

A POD consists of three separate ASCII text files. The name of each POD file is formed from the printer name, followed by one of these suffixes: *.config*, *.status*, or *.log*. The name and contents of each file are as follows:

[*printer_name*].*config*   The configuration file contains detailed information on the printer's capabilities. Examples include the supported paper sizes and available fonts. The initial version of this file is manually created by the printer driver developer, and a copy is installed when the printer is added to the system. The contents of this file are maintained by the system administrator using the printer administration tools. Normally this file is never changed.

[*printer_name*].*status*   The status file contains information about the current operational status of the printer. The information in this file indicates whether the printer is busy, the type of printing media installed, detailed error codes (if errors have occurred), and so on. The contents of this file change during the course of every print job.

[*printer_name*].*log*   The log file contains the print job history for the printer. Information for old jobs as well as the current print job is maintained. Typically, printer filters and drivers append information to the log file while general applications treat the file as read-only.

The global variable *PDpod_path* indicates the location of the POD files. The default location for the POD files is */var/spool/lp/pod*. If the POD files are to be located in a directory other than the default, set *PDpod_path* to the path name of the new location. *PDpod_path* is declared in the header file *pod.h*.[1]

See Appendix C, "Printer Object Database (POD) File Formats," for detailed information on *libpod* file formats.

Standard and Local libpod Functions

The *print server* is the system that controls the printer. The POD files reside on the print server. To provide POD information to a system other than the print server, that is, to a *print client, libpod* must be able to communicate across the network. For any specified printer, the "standard" *libpod* functions automatically determine whether the user's system is the print server or a print client. After determining which system is the print server, the standard *libpod* functions are able to access the POD files on that system. Because they typically run on print clients, user application programs such as printer status tools use the standard form of the *libpod* functions.

To avoid the overhead that network communication entails, *libpod* also provides "local" functions. These functions, such as **PDLocalReadInfo()** and **PDLocalWriteStatus()**, contain the word "Local" in their names and use the POD files on the system on which they are running, that is, the local system. Because they have no networking overhead, their use reduces the size and overhead of the resulting executable files. They are intended for programs, such as printer drivers, that are used only on print servers.

Functions that write to POD files are available only in the local form. Only printer drivers write to POD files and these drivers always run locally on print servers, never remotely on print clients. Thus, there is no need to provide standard *libpod* functions that write to POD files.

---

[1] The maximum string length for *PDpod_path* is PD_STR_MAX. This length includes the terminating NULL character.

Compiling Programs With libpod

Programs that call *libpod* functions must include the header file *pod.h*, which is located in the directory */usr/include*. The programs must also link with the library *libpod.a* located in */usr/lib*. In addition, programs that use the standard *libpod* functions must link with *libspool.a*. Programs that use only the local *libpod* functions need not link with this additional library.

The compile line for using the standard functions is

```
cc -o myprog myprog.c -lpod -lspool
```

The compile line for using the local functions is

```
cc -o myprog myprog.c -lpod
```

Debugging With libpod

If the global variable *PDdebug* is set to a nonzero value, *libpod* functions will print debugging information to standard error during execution. The global variable *PDdebug* is declared in the header file *pod.h*.

Network Communications

To provide remote printer POD information, *libpod* communicates over the network with the *podd* daemon on the remote machine. Since the network or remote machine may be unreachable when a *libpod* function is executed, a time-out may occur with the function returning an appropriate error code. The time-out period can be specified by setting the global variable *PDnet_timeout* to a value in seconds. The default time-out period is contained in the header file *pod.h*. The time-out period for reading printer status is usually much larger than that for browsing printer configurations. For more specific information on the daemon, refer to the podd(1M) reference page.

Library Functions Listed by Purpose

The *libpod* functions are listed in Table 5-3. Note that a number of *libpod* functions have only a single version, which is used for both standard and local cases.

Table 5-3   Summary of libpod Functions

Task	Standard Function	Local Function
Detailed Information Retrieval	PDReadInfo()	PDLocalReadInfo() PDLocalWriteInfo()
Status File Manipulation	PDReadStatus()	PDLocalReadStatus() PDLocalWriteStatus()
	PDReadOpStatus()	PDLocalReadOpStatus()
Log File Manipulation	PDReadLog()	PDLocalReadLog() PDLocalWriteLog()
Convenience Functions	PDMakeMessage()	n/a
	PDFindPageSize()	n/a
	PDGetSizeCodeByName()	n/a
	PDGetNameBySizeCode()	n/a
	PDGetCurrentResolution()	n/a
Execution Error Handling	PDError()	n/a
	PDErrorString()	n/a

Scanner Drivers

This chapter discusses scanner driver development. It provides a detailed analysis of the template scanner driver.

Scanner Drivers

This chapter discusses scanner driver development. It provides a detailed analysis of the template scanner driver.

The following major topics are discussed in this chapter:

- "Driver Template" on page 72
- "Header Files" on page 72
- "Data Structures" on page 73
- "Functions You Must Write" on page 78
- "Events" on page 88
- "Installation" on page 90
- "Testing" on page 91

The information presented in this chapter should be enough to write a scanner driver. However, if you wish to know more, Appendix E, "Scanner Driver Architecture," is an in-depth discussion of the architecture of a scanner driver.

Driver Template

The source code files for the template scanner driver are in the directory
/usr/impressario/src/scan/template_driver. This template has code to handle all the
interprocess communication necessary for well-behaved scanner drivers (see Chapter 8,
"Generic Scanner Interface").

To develop a new scanner driver, start by copying the template files to the directory
where you will be developing the driver. The *only* files that you need to modify are *scan.c*
and *Makefile*; *Do not* modify any of the other files unless you are familiar with the
information in Appendix E.

This document refers to the *main.c* module, which implements the interprocess
communication part of a scanner driver and should not be modified; and *scan.c*, which
you should modify to support the scanner for which you are writing a driver.

Header Files

There are four header files in */usr/include* that are useful to scanner driver developers:

scanner.h   Defines the interface used by application programmers to communicate
with scanner drivers. Contains **typedef** and **#define** definitions needed
to communicate with the application.

scandrv.h   Contains the dispatch loop interface, some error messages, and the
queue utility routines.

scanipc.h   Contains **#define** definitions for command numbers and the types of
arguments and results.

scanconv.h   Contains prototypes for functions that convert between data types and
functions that do replicative zooming on rows of image data.

Data Structures

The following data structures are used to communicate between scanner driver template modules. Understanding each field is key to understanding what your part of the driver (the code in *scan.c*) must do. These data structures are defined in */usr/impressario/src/scan/template_driver/scan.h.*

SCANINFO Data Structure

The **SCANINFO** data structure is used to store static information about a scanner. The *scan.c* module uses **SCANINFO** to communicate with the *main.c* module. The **SCANINFO** data structure is defined as follows:

```
typedef struct tag_scaninfo {
    int metric;                                 /* metric for res, page size*/
    float pagex, pagey, pagewidth, pageheight;  /* page size */
    float minxres, maxxres, minyres, maxyres;   /* resolution bounds */
    float *xres, *yres;                         /* resolution arrays */
    int nres;                                   /* size of arrays */
    int canZoom;                                /* 1 if scanner can zoom */
    SCDATATYPE *types;                          /* supported types */
    int ntypes;                                 /* number of types */
    SCANFUNC *options;                          /* scanner-specific options */
    int noptions;                               /* number of options */
    void *priv;                                 /* private member */
    SCFEEDERFLAGS feederFlags;                  /* feeder flags */
} SCANINFO;
```

Field definitions:

metric Set *metric* to SC_INCHES or SC_CENTIM, depending on whether it is more convenient to have measurements and resolutions expressed in terms of inches or centimeters.

pagex, pagey, pagewidth, pageheight
  pagex and *pagey* are the coordinates of the upper left corner of the scannable area (almost always 0 and 0). *pagewidth* and *pageheight* are the width and height of the scannable area, respectively. All four fields are expressed in the units defined by the *metric* field.

minxres, maxxres, minyres, maxyres

        minxres is the smallest supported horizontal resolution, *maxxres* is the largest supported horizontal resolution, *minyres* is the smallest supported vertical resolution, and *maxyres* is the largest supported vertical resolution. These are all expressed in pixels per the unit expressed by the *metric* field.

xres, yres, nres   For scanners that support discrete resolutions (as opposed to scanners that support all resolutions with equal quality, within the bounds given above), *xres* and *yres* are arrays of the supported resolutions in the horizontal and vertical directions. *nres* is the number of elements in each of these arrays.

        For scanners that support arbitrary resolutions (that is, scanners that do their own scaling), *nres* is 0. The *main.c* module takes *nres* equal to 0 to signify that it doesn't need to do any scaling of scan data to satisfy preview requests from the scanning application.

canZoom   This parameter specifies whether or not the scanner can support resolutions other than those specified in the *xres* and *yres* arrays when *nres* is nonzero. In this case, the resolutions in the *xres* and *yres* arrays represent preferred resolutions that results in superior image quality.

        If *nres* is 0, the *main.c* module assumes that the scanner itself can do zooming, regardless of the *canZoom* flag.

types, ntypes   *types* is an array of **SCDATATYPE** structures (see Chapter 8, "Generic Scanner Interface"), and *ntypes* is the number of types supported by the scanner.

options, noptions

        This array of functions implements scanner-specific options for this scanner (see Chapter 7, "Scanner-Specific Options"), and *noptions* is the number of such options.

priv   This parameter is used by the *scan.c* module (the one you write) to store a pointer to whatever state information is necessary to identify a particular scanner once it's been opened. This is provided so that you can avoid the use of global variables in the *scan.c* module.

feederFlags   These flags indicate the presence of an automatic document feeder. The SC_HASFEEDER bit (see */usr/include/scanner.h*) of this flag should be set if a feeder is attached to the scanner being supported. The SC_AUTOFEED flag should be set if each call to **DoScan()** automatically results in the next sheet of paper being fed. If the scanner can feed on demand, the SC_PROGFEED bit should be set. It is not an error to have the SC_AUTOFEED flag and the SC_PROGFEED flag both set.

If the scanner being supported does not have a document feeder, this member can be safely ignored and the *main.c* module will not try to call any of the document feeder functions (see below).

SCANPARAMS Data Structure

The **SCANPARAMS** data structure contains dynamic values used to specify the parameters of a scanning operation, and also some administrative details. The **SCANPARAMS** data structure is defined as follows:

```
typedef struct tag_scanparams {
    float xres, yres;
    float x, y, width, height;
    SCDATATYPE type;
    int preview;
    SCQUEUE *scanq, *sfreeq;
    int xpixels, ylines, xbytes;

    void (*convert)(void *from, int fromx, void *to, int tox, int *zmap);

    int maxmem; /* maximum amount of memory to allocate */
    int readlines;
    SCANINFO *s;
} SCANPARAMS;
```

The fields of the **SCANPARAMS** data structure are defined as follows:

xres, yres　　The scanning resolution to be used for a particular scan. The *main.c* module always ensures that these resolutions are among those advertised in the *xres* and *yres* fields of the **SCANINFO** struct, unless the *canZoom* field of the **SCANINFO** struct is nonzero or the *nres* field is 0. In any case, *xres* and *yres are* always within the resolution bounds specified in the **SCANINFO** struct.

　　　　　　　　xres and *yres* are expressed in dots per the unit specified in the *metric* field of the **SCANINFO** struct.

x, y, width, height
　　　　　　　　The horizontal (*x*) and vertical (*y*) coordinates of the upper left corner of the window to be scanned and its width and height. The *main.c* module ensures that this image falls within the bounds of the *pagex, pagey, pagewidth*, and *pageheight* fields of the **SCANINFO** struct.

　　　　　　　　x, y, width, and *height* are expressed in the units specified in the *metric* field of the **SCANINFO** struct.

type　　　　The type of scan data expected. The *main.c* module ensures that it is one of the types specified in the *types* field of the **SCANINFO** struct.

preview　　This field is set to 1 if this is a preview scan, and 0 otherwise.

scanq, sfreeq　　*sfreeq* is a queue whose elements are free buffers to put scanned data into, and *scanq* is a queue whose elements are buffers that have scanned data in them. **DoScan()**, which you write (see below), removes buffers from *sfreeq*, scans the data into them, then adds them to *scanq*. The *main.c* module is responsible for taking buffers from *scanq*, disposing of the data appropriately, and putting them back on *sfreeq*.

xpixels, ylines, xbytes
　　　　　　　　The number of pixels in a scan line, the number of scan lines in the scan, and the number of bytes in a scan line. The *scan.c* module is responsible for calculating these values in **SetupScan()**, which you write (see below).

void (***convert**)(void *from, int *fromx*, void *to, int *tox*, int *zmap)

> This function converts data from a type that the scanner supports to the requested data type. If the scanner directly supports all the data types that are being advertised to the scanning application (the *types* field of the **SCANINFO** struct), the *scan.c* module can ignore this field.
>
> For example, this function can be used for color scanners that return the red, green, and blue components of each scan line separately; that is, a line of five pixels would have the following layout:
>
> RRRRRGGGGGBBBBB
>
> This needs to be converted to *chunky* data, as shown below:
>
> RGBRGBRGBRGBRGB
>
> To do this, simply set the *convert* field to *SCBandRGB8ToPixelRGB8* in **SetupScan()** (see below). The following functions are available in *libscan* for converting:
>
> • **SCBandRGB8ToPixelRGB8()**
>
> • **SCGrey8ToMono()**
>
> • **SCBandRGB8ToMono()**

maxmem
> The maximum amount of memory that should be allocated for storing scan data. This field is to be taken into account in the calculation of *readlines* in **SetupScan()** (see below).

readlines
> The number of lines to read at a time. *readlines* is the *maxmem* field divided by the *xbytes* field if scanning is benefited by scanning in large chunks. If there is no benefit, the number is 1.
>
> The problem with *maxmem/xbytes* is that when *maxmem* is large, interactive feedback to the user of the scanning application is limited. Ideally, the scanner buffers data internally, so you can scan perhaps an inch at a time without the scan head pausing. That way, the scanning application can consume the scan data while the scan head gets the rest of the data.

s
> A pointer to the **SCANINFO** struct that **OpenScanner()** returned (see below).

Functions You Must Write

After copying the template to your build area, you must edit the file *scan.c* and implement the functions listed in Table 6-1.

These functions are described in detail in the following sections.

Table 6-1   Functions to Be Written by the Driver Developer

Function Name	Description
OpenScanner()	Opens the scanner
SetupScan()	Called before a scanning operation
DoScan()	Gets data from the scanner
SetFeederFlags()	Called when the scanner application calls SCFeederSetFlags
AdvanceFeeder()	Advances feeder to next document
FeederReady()	Tests whether the feeder is ready to feed another document
PrintID()	Prints a string describing the type of scanning supported
FindScanners()	Prints device for supported scanners
InstallScanner()	Installs a new scanner
DeleteScanner()	Deletes a scanner

OpenScanner() Function

This function is called when the driver is first invoked.

For example:

```
SCANINFO *
OpenScanner(char *dev)
{
    static SCANINFO scan;

    /*
    Your code here!
    */

    if (something goes wrong) {
        drverr = appropriate error code;
        return NULL;
    }
    */

    return &scan;
}
```

dev is the name of the device (usually a device special file in */dev/scsi* for SCSI devices) to open in order to communicate with the scanner. The task of the **OpenScanner()** function is to "open" *dev*, make sure that it corresponds to a device that *scan.c* knows how to talk to, get it into some reasonable initial state, and fill in a **SCANINFO** structure for the scanner. If all goes well, a pointer to the **SCANINFO** structure is returned.

If anything goes wrong, **OpenScanner()** should set the global variable *drverr* and return NULL. The value for *drverr* should be chosen from those in */usr/include/sys/errno.h* or */usr/include/scanner.h*; that value is communicated back to the scanning application, which can use the **SCPerror()** or **SCErrorString()** functions in *libscan.a* to get a human-readable error message that explains why **OpenScanner()** failed.

Caution: If you are writing a driver for a SCSI scanner, and you are using dslib(3X), make sure that you pass the O_EXCL flag defined in */usr/include/fcntl.h* to *dsopen*:

```
dsreq_t *dsp = dsopen(dev, O_RDONLY | O_EXCL);
```

If you pass the O_EXCL flag, the open will fail with *errno* set to EBUSY if *dev* is the */dev/scsi* device of a mounted disk; otherwise, the open can succeed and you could really screw up the disk!

79

In addition, it is recommended that before issuing any other SCSI commands you perform an inquiry command, and verify that the device is a scanner by examining the Device Type code of the inquiry buffer. (This field should be set to *6*. You can use the INV_SCANNER **#define** from */usr/include/invent.h*.) It is also recommended that you examine the vendor and product identifiers to make sure the device is a scanner of the type for which this driver is being written.

SetupScan() Function

This function is called with a pointer to a **SCANPARAMS** struct to prepare for a scanning operation.

For example:

```
int
SetupScan(SCANPARAMS *params)
{
    /*
    Your code to tell the scanner the resolution, scanning window,
    and data type.
    */

    /*
    Your code to find out from the scanner how many pixels are in a scan
    line, how many scan lines are in the scan, and how many bytes are
    in a scan line.
    */

    /*
    Your code to figure out what readlines should be, taking into
    consideration maxmem and xbytes.
    */

    if (anything went wrong) {
        drverr = an appropriate error code;
        return -1; /* indicates failure */
    }

    return 0; /* indicates success */
}
```

SetupScan() performs the following operations:

1. **SetupScan()** does whatever is necessary to anticipate the scan defined by the fields *xres, yres, x, y, width, height*, and *type.*

2. If *type* is not supported directly by the scanning device, then the *convert* field should be set to a function that converts data returned from the scanner to the appropriate type.

3. **SetupScan()** queries the scanning device or does some calculations to determine the number of pixels in a scan line, the number of scan lines in the scan, and the number of bytes in a scan line. The *xpixels, ylines*, and *xbytes* fields of *params* are set appropriately.

4. **SetupScan()** sets the *readlines* field of *params* to the number of lines that it expects to scan at a time, taking *maxmem* and *xbytes* into account.

5. If at any point something goes wrong, set *drverr* to a value from */usr/include/sys/errno.h* or */usr/include/scanner.h* and return -1 to indicate a failure. If all goes well, return 0 to indicate success.

DoScan() Function

This function retrieves the data from the scanner.

For example:

```
void
DoScan(SCANPARAMS *params)
{
    SCANINFO *s = params->s;
    void *buf;
    int row, toread, curline;

    prctl(PR_TERMCHILD);

    for (curline = 0; curline < params->ylines;
        curline += params->readlines) {
        toread  = MIN(params->readlines,
                    params->ylines - curline);

        buf = SCDequeue(params->sfreeq);
```

```
            /*
             * Get the scan data here!
             */

            /*
             * Chop the buffer up into scan line sized chunks
             */
            while (toread--) {
                SCEnqueue(params->scanq, buf);
                buf = (char *)buf + params->xbytes;
            }
        }

    exit(0);
}
```

DoScan() executes as its own process, sharing its address space with its parent, which is the process that communicates with the scanning application. (See the sproc(2) reference page. **DoScan()** is the *entry* parameter to *sproc*.)

Before entering the while loop, do whatever else is necessary to initialize the scanner if there's unfinished business from **SetupScan()**. Note the use of *params->readlines*, which you set in **SetupScan()**. In the body of the loop, the following things happen:

1. **DoScan()** computes how many scan lines to read this time through the loop. This is either *readlines*, which was set in **SetupScan()**, or the number of lines remaining to scan:

    ```
    toread = MIN(params->readlines,
                 params->ylines - curline);
    ```

2. **DoScan()** gets a buffer from the free queue into which the data is scanned:

    ```
    buf = SCDequeue(params->sfreeq);
    ```

3. **DoScan()** transfers *toread* lines of data from the scanning device to *buf*. This is the interesting part, that you have to write specifically for your scanner.

4. **DoScan()** puts the lines just scanned onto the *scan* queue. This involves chopping up the buffer into chunks the size of a scan line. Don't worry, *main.c* knows how to put the buffers back together before putting them back on the free queue!

    ```
    while (toread--) {
        SCEnqueue(params->scanq, buf);
        buf = (char *)buf + params->xbytes;
    }
    ```

Since **DoScan()** is its own process, it calls the *exit* function instead of returning when it finishes scanning. If everything goes OK, **DoScan()** calls *exit* with a status of 0. If anything goes wrong, **DoScan()** sets the global variable *drverr* to an appropriate value from *sys/errno.h* or *scanner.h* and calls *exit* with a status of 1. (See the exit(2) reference page.)

SetFeederFlags() Function

The **SetFeederFlags()** function is called when the scanner application calls *SCFeederSetFlags* to specify whether automatic (SC_AUTOFEED) or programmatic (SC_PROGFEED) feeding is desired. This only happens if the *feederFlags* member of the **SCANINFO** struct returned by **OpenScanner()** has all three of the SC_HASFEEDER, SC_AUTOFEED, and SC_PROGFEED bits set.

For example:

```
int
SetFeederFlags(SCANINFO *scan, SCFEEDERFLAGS flags)
{
    drverr = SCENOFEEDER;
    return -1;
}
```

The template version of this function sets *drverr* to indicate that no feeder is present; if a feeder is present, **SetFeederFlags()** must set a flag so that it knows whether to automatically feed the next document in the next call to **DoScan()**.

AdvanceFeeder() Function

The **AdvanceFeeder()** function is called only if the SC_PROGFEED bit is set in the *feederFlags* member of the **SCANINFO** struct returned by **OpenScanner()**. This function should advance the feeder to the next document. If the feeder is empty or jammed, return -1 and set *drverr* to an appropriate error code from */usr/include/scanner.h* or */usr/include/sys/errno.h.*

For example:

```
int
AdvanceFeeder(SCANINFO *scan)
{
    drverr = SCENOFEEDER;
    return -1;
}
```

FeederReady() Function

This function is called only if the SC_HASFEEDER bit of the *feederFlags* field of the
SCANINFO struct returned by **OpenScanner()** is set.

For example:

```
int
FeederReady(SCANINFO *scan)
{
    drverr = SCENOFEEDER;
    return -1;
}
```

FeederReady() should return 0 if there is a document in the feeder; that is, if the next call
to **AdvanceFeeder()** should succeed. If the feeder is empty, **FeederReady()** should return
-1 and set *drverr* to SCFEEDEREMPTY.

PrintID() Function

The **PrintID()** function is used by the -**query** option that all scanner drivers support. It
should print a string that identifies the type of scanner supported by this scanner driver,
and one or more interface types supported. The scanner install tool, scanners, uses this
information to help the end user choose the driver that best suits a particular scanner.

For example:

```
void
PrintID(FILE *fp)
{
    fprintf(fp, "Your Scanner Name\n");   /* String describing scanner */
    fprintf(fp, "SCSI Serial Parallel\n"); /* Device type; can be list */
}
```

FindScanners() Function

The **FindScanners()** function is also used to implement the -**query** option.

For example:

```c
void
FindScanners(FILE *fp)
{
    inventory_t *inv;
    char device[100];
    dsreq_t *dsp;
    /* int because it must be word aligned. */
    int inqbuf[(sizeof(INQDATA) + 3)/sizeof(int)];
    INQDATA *inq = (INQDATA *)inqbuf;

    /*
     * This example looks for SCSI scanners; do whatever is necessary
     * to find other types of scanner here.
     */
    setinvent();
    while ((inv = getinvent()) != NULL) {
        if (inv->inv_class == INV_SCSI && inv->inv_type == INV_SCANNER) {
            sprintf(device, "/dev/scsi/sc%dd%dl0", inv->inv_controller,
                inv->inv_unit);
            if ((dsp = dsopen(device, O_RDONLY)) == NULL) {
                continue;
            }

            if (inquiry12(dsp, (char *)inq, sizeof *inq, 0) == 0
                && strncmp((char *)inq->vid, "Your vendor", 11) == 0 &&
                strncmp((char *)inq->pid, "Your product", 12) == 0) {
                    fprintf(fp, "SCSI %s\n", device);
            }
            dsclose(dsp);
        }
    }

    endinvent();
}
```

FindScanners() should search the system for scanners that this driver is capable of supporting, and for each such scanner it prints the type of device (SCSI, Serial, Parallel, GPIB, EISA, or Other), a space, and the pathname that should be passed to **OpenScanner()** in order to access that scanner.

This gives scanners more information that it can use to help the end user pick a driver for a particular scanner. It is by no means required that **FindScanners()** find all scanners that it is capable of supporting; it is OK to do nothing at all here, especially if there is no hope of finding a scanner you support in a reasonable amount of time. This is important; scanners invokes EVERY scanner driver installed with the -**query** option when the user adds a scanner, so this function should be fast!

InstallScanner() Function

This function is called when the scanner driver is invoked with the -**install** option. **InstallScanner()** is used by scanners when the user tries to install a new scanner.

For example:

```
int
InstallScanner(char *dev)
{
   printf("The template driver doesn't support %s\n", dev);
   return -1;
}
```

The purpose of this entry point is verify that *dev* corresponds to a scanning device that this driver knows how to support, and to do any scanner-specific installation that is necessary.

The following example implementation calls **OpenScanner()** to verify that *dev* corresponds to a valid scanning device, and then changes the permissions of *dev* so that users other than root can access the scanner. This is important, because scanner drivers should not normally be set to user ID *root* programs, and users other than root want to use scanners. When **InstallScanner()** is called by scanners, the driver will have root permissions, which enables it to call chmod(2) on *dev* or create any auxiliary files or other resources that it needs:

```
int
InstallScanner(char *dev)
{
    SCANINFO *scan;

    scan = OpenScanner(dev);

    if (!scan) {
        printf("Can't access %s: %s\n", dev,
                SCErrorString(drverr));
        return -1;
    }

    chmod(dev, 0666);
    return 0;
}
```

If an error occurs, **InstallScanner()** should print an error message to standard out and return -1. The *main.c* module exits with a nonzero exit status if **InstallScanner()** returns -1, and scanners reads the driver's standard output and displays it to the user if the *main.c* exits with a nonzero status. That way, the exact cause of the error is propagated to the user.

DeleteScanner() Function

The **DeleteScanner()** function is called when the driver is invoked with the -**delete** option by scanners. This gives the driver the opportunity to do any scanner-specific deletion required. This can be useful if auxiliary files specific to this scanner were created in **InstallScanner()**.

For example:

```
int
DeleteScanner(char *dev)
{
    return 0;
}
```

If an error occurs, **DeleteScanner()** prints an error message to standard out and returns -1. In this case, scanners displays this error message to the user and refuses to delete the scanner, so in most cases **DeleteScanner()** should return 0.

Events

Impressario scanner drivers can send events to scanner applications. Currently, the only type of event supported is an event to notify the scanner application that the resolutions, page size, data types, or feeder flags supported by the scanner driver have changed.

This typically happens when the user selects a new input media option from the scanner specific options program (see chapter 8). For example, some scanners have transparency units, and when scanning transparencies the scanning page size is different than when scanning normal paper. So when the user selects the option to scan a transparency, the scanner driver needs to inform the scanning application that it must query to find out the new page size.

Events are sent to the scanner application by filling in an **SCEVENT** structure and calling **SCDriverSendEvent()**. The **SCEVENT** structure is defined as follows:

```
typedef struct tag_infoChange {
    unsigned int pageSizeChanged : 1;
    unsigned int resolutionChanged : 1;
    unsigned int dataTypesChanged : 1;
    unsigned int feederFlagsChanged : 1;
} SCINFOCHANGE;

#define SCEVENT_INFOCHANGE 1

typedef struct tag_scevent {
    unsigned int eventType;
    union {
        SCINFOCHANGE infoChange;
    event;
} SCEVENT;
```

The **SCDriverSendEvent()** function has the following prototype:

```
int SCDriverSendEvent(SCEVENT *event)
```

To inform the scanner application that it needs to query the new page size, the driver executes the following code:

```
SCEVENT event;

event.eventType = SCEVENT_INFOCHANGE;
event.event.infoChange.pageSizeChanged = 1;
event.event.infoChange.resolutionChanged = 0;
event.event.infoChange.dataTypesChanged = 0;
event.event.infoChange.feederFlagsChanged = 0;
if (SCDriverSendEvent(&event) == -1) {
    handle error;
}
```

Installation

After the driver is built, make sure that the -**query** option works, then copy it to */usr/lib/scan/drv* and run *scanners*, the scanner install tool. See Figure 6-1.
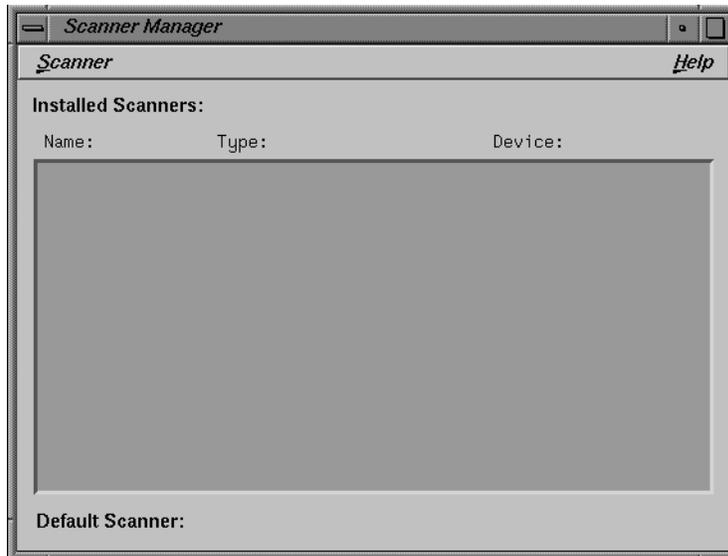
Figure 6-1   Scanner Install Tool

When the *scanners* panel comes up, use the "Install…" item on the Scanner menu to bring up the "Install New Scanner" dialog box. If you implemented **PrintID()** correctly, you see your scanner driver's ID string in the list. If you implemented **FindScanners()**, clicking on your scanner driver type should fill in the "Device" field. Otherwise, type in the device that corresponds to your scanner.

Testing

Give the scanner a name and click *OK*, then run *gscan*. See Figure 6-2.
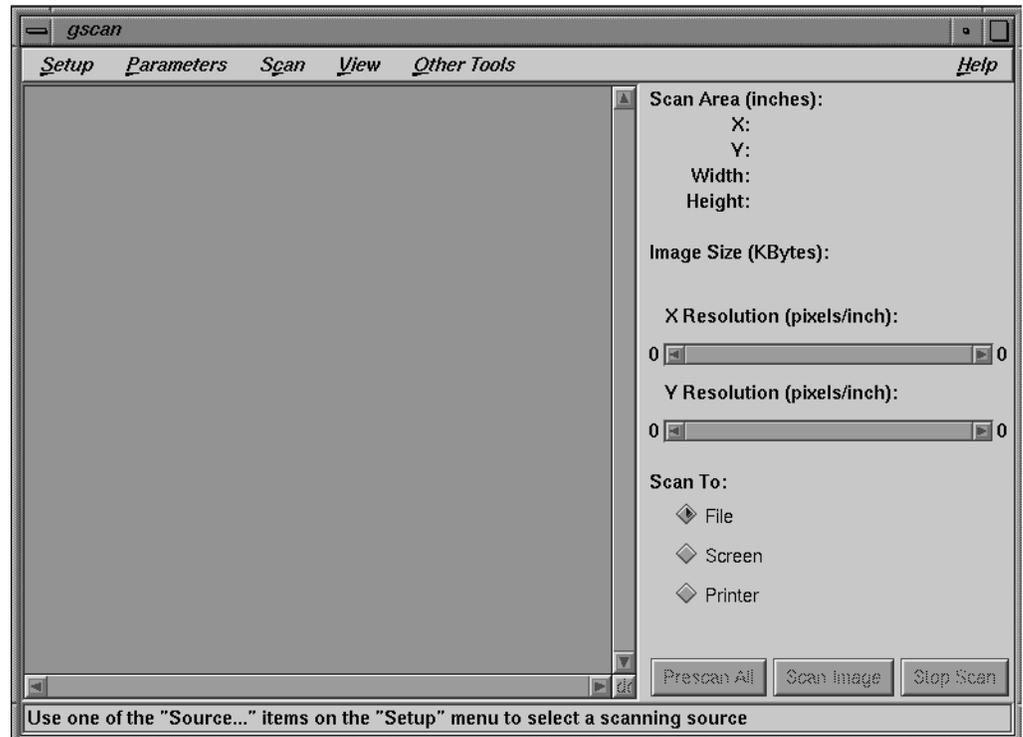
Figure 6-2   gscan Panel

You can either provide your scanner's name on the command line or use the Setup menu to choose your scanner as the scanning source.

Severe scanner driver malfunctions can cause *gscan* to hang. If this happens, open a shell and enter:

```
/etc/killall gscan
```

Scanner-specific Options

This chapter explains how to implement a scanner-specific graphical options panel. Doing so allows you to showcase your scanner's features and enhance the overall usability and visual appeal of your product.

Scanner-Specific Options

This chapter describes the implementation of scanner-specific graphical options panels.

The following major topics are discussed in this chapter:

Overview

Most scanners have capabilities that would not be available to application programs through the generic scanner API alone. A scanner-specific graphical options panel program can be developed to provide access to these capabilities.

A scanner-specific options program is run by a scanner application and communicates directly with a scanner driver, bypassing the generic scanner API. It should provide user interface elements (using Motif or a similar toolkit) for the scanner features it supports that are not supported by the generic scanner API. It should also communicate appropriate settings to the scanner driver.

The scanner driver, in turn, must respond to commands from its corresponding scanner-specific options program.

Options Program and the Scanner Driver Interface

The options program executes driver commands by writing a command number and the arguments to that command onto a pipe. The scanner driver reads the command and arguments from the pipe, then calls the options function for that command. After the options function returns, the driver writes the results of the function back to the options program.

Developers of new scanner drivers and options programs need not worry about the low-level communication between these programs; this is all taken care of in *libscan.a*. They *do* need to ensure that the scanner drivers and options programs interpret commands and arguments consistently. This is most easily accomplished by providing a common header file that the driver and the scanner options program share. This header file should contain **#defines** for the scanner-specific commands and **typedefs** for the arguments, and return values of these commands. *sclopt.h*, for example, is included by both the HP ScanJet scanner driver and the HP ScanJet options panel:

```
/*
 * sclopt.h
 * Stuff for scl scanner-specific options
 */

#define SCL_GETOPTS (SCN_SCANSPECIFIC + 0)
#define SCL_SETOPTS (SCN_SCANSPECIFIC + 1)

#define DITH_COURSE 0
#define DITH_FINE 1
#define DITH_BAYER 2
#define DITH_VERTICAL 3

typedef struct tag_sclopt {
    int intensity;                  /* Intensity of image */
    int minIntensity, maxIntensity; /* Intensity bounds; used */
                                    /*  in GETOPTS only */
    int contrast;                   /* Image contrast */
    int minContrast, maxContrast;   /* Contrast bounds; used */
                                    /*  in GETOPTS only */
    int bwDither;                   /* if nonzero, dither  */
                                    /* black and white data */
    int bwDitherPattern;            /* specify black & white */
                                    /* dither pattern */
} SCLOPT;
```

sclopt.h defines two HP ScanJet-specific commands: *SCL_GETOPTS* and *SCL_SETOPTS*. Scanner-specific commands *must* be numbered consecutively, starting with *SCN_SCANSPECIFIC* and increasing monotonically from there. *SCN_SCANSPECIFIC* is defined in */usr/include/scanipc.h*.

The **SCLOPT** structure is used to pass information between the HP ScanJet scanner driver and the HP ScanJet options panel. This structure is the return type of the *SCL_GETOPTS* command and the argument type of the *SCL_SETOPTS* command.

"Scanner Driver's Perspective" below describes how the scanner driver uses these **#defines** and **typedefs**. "Options Program's Perspective" on page 100 describes how the scanner-specific options program uses them.

Scanner Driver's Perspective

The scanner driver implements scanner-specific options by providing a table of functions to the *main.c* module. In **OpenScanner()**, the *scan.c* module should set the *options* field of the **SCANINFO** struct to an array of functions implementing the scanner-specific options, and the *noptions* field to the number of such options. The order of the functions in the options table must correspond to the numerical order of the commands implemented (the *cmd* argument, below).

Each function in this array must have the following prototype:

```
void optionfunc(int cmd, SCARG *arg, SCRES *res);
```

cmd is the command number for this scanner-specific option. SCARG is defined in the file */usr/include/scandrv.h* as follows:

```
typedef struct tag_scarg {
    void *data;
    int len;
} SCARG;
```

arg->data points to the arguments passed in by the scanner-specific options program, and *arg->len* is the number of bytes pointed to by *arg->data*.

SCRES is defined in the file */usr/include/scandrv.h* as follows:

```
typedef struct tag_scres {
   void *data;
   int len;
   void *freeparam;
   void (*free)(void *param, void *data);
   int errno;
   char *errmsg;
} SCRES;
```

res->data should be set to point to the results of the scanner-specific option, and *res->len* should be set to point to the number of bytes in *res->data*. If an error occurs, *res->errno* should be set to one of the values from *sys/errno.h* or *scanner.h*. If *res->free* is nonzero, it is called with *res->freeparam* as its first argument and *res->data* as its second argument, after *res->data* has been transferred to the scanner-specific options program. This can be used to free memory that was dynamically allocated to temporarily hold the results.

For example, here is the code from the ScanJet driver that implements the HP ScanJet options:

```
static SCLOPT scanOptions;

static void
GetOptions(int cmd, SCARG *arg, SCRES *res)
{
   res->data = &scanOptions;
   res->len = sizeof scanOptions;
}

static void
SetOptions(int cmd, SCARG *arg, SCRES *res)
{
   scanOptions = *(SCLOPT *)arg->data;
}

SCANFUNC opttable[] = {
   GetOptions,
   SetOptions,
};
```

In **OpenScanner()**, the *options* member of the **SCANINFO** struct is set to *opttable*, and the *noptions* member is set to 2. Note that the order of the functions in *opttable* corresponds to the numerical order of the commands they implement.

SCANFUNC is a **typedef** from */usr/include/scandrv.h*:

```
typedef void (*SCANFUNC)(int cmd, SCARG *arg, SCRES *res);
```

Note: The actual code in the ScanJet driver is slightly more complex; error checking has been eliminated from this example code so as not to obscure the basic functionality.

The **SetOptions()** function should verify that the options passed to it are valid and set *res->errno* to a value from the */usr/include/sys/errno.h* directory or */usr/include/scanner.h* if they are not. Also, the ScanJet function **SetOptions()** will fail if the scanner is currently scanning, because verifying that the options are valid would interrupt the scan. So, in this case, **SetOptions()** should set *res->errno* to SCEBUSY and return.

Options Program's Perspective

The scanner-specific options program is executed by *libscan.a* when the scanner application calls the function **SCOptions()**. The program is executed with command-line arguments that, when passed to the *libscan.a* function **SCGetScanOpt()**, enable a connection to be established with the scanner driver.

One of the first things that a scanner-specific options program must do, then, is call **SCGetScanOpt()**. This function has the following prototype (from */usr/include/scanipc.h*):

```
SCANOPT * SCGetScanOpt(int *argc, char *argv[]);
```

The scanner-specific options program communicates with the scanner driver by making calls to the function **SCScanOpt(),** which has the following prototype (from */usr/include/scanipc.h*):

```
int SCScanOpt(SCANOPT *s, int cmd, void *args, int arglen,
    void *res, int reslen);
```

The first argument to **SCScanOpt()** is the pointer returned by **SCGetScanOpt()**, above. The *cmd* argument is one of the command **#defines** from the common header file shared with the scanner driver. *args* is a pointer to the arguments to this command, and *arglen* is the number of bytes pointed to by *args*. This corresponds to *arg->data* and *arg->len* in the scanner driver option function for *cmd*.

res points to space for receiving the results of the command, and *reslen* is the maximum number of bytes to copy into *res*. The data copied into *res* corresponds to *res->data* in the scanner driver option function for *cmd*.

The following example code was distilled from the ScanJet options program. The ScanJet options program is a Motif program; please refer to the X and Xt Motif documentation set (see "Related Publications," in the "About This Guide" section of this manual for the full names and order numbers) for information about the non-scanning portions of the code below.

```
static Widget toplevel;
static SCANOPT *scan
static SCLOPT scanOptions;
static XtAppContext appContext;

int
main(int argc, char *argv[])
{
    toplevel = XtAppInitialize(&appContext, "SJIIcOpt", NULL, 0,
                (unsigned int *)&argc, argv, fallBackResources,
                NULL, 0);

    scan = SCGetScanOpt(&argc, argv);

    if (!scan) {
        InitError(toplevel, appContext,SCErrorString(SCerrno));
    }

    if (SCScanOpt(scan, SCL_GETOPTS, NULL, 0,
        &scanOptions, sizeof scanOptions) < 0) {
        InitError(toplevel, appContext, SCErrorString(SCerrno));
    }

     /* ... create widgets corresponding to options ... */

    XtRealizeWidget(toplevel);
    XtAppMainLoop(appContext);
    return 0;
}
```

main() calls XtAppInitialize(3Xt) to initialize the X toolkit, and then calls **SCGetScanOpt()** to get the connection to the scanner driver. Then it calls **SCScanOpt()** to get the current options settings from the scanner driver.

If anything goes wrong, **main()** calls a function (not shown here, but part of the ScanJet options program) called **InitError()**, which transforms the application into a message dialog containing the error message passed as the third argument.

The ScanJet options program has an *OK* button that the user presses after setting up the options desired. Below is an excerpt from the callback function for that *OK* button.

```
static void
OKCallback(Widget w, XtPointer client,
           XmAnyCallbackStruct *cb)
{
   /* ... get settings from widgets,
          put them into scanOptions
   ... */

   if (SCScanOpt(scan, SCL_SETOPTS, &scanOptions,
       sizeof scanOptions, NULL, 0) < 0) {
           PostError(toplevel, SCErrorString(SCerrno), 0);
           return;
   }
   exit(0);
}
```

Again note the error check; **PostError()** is another function from the ScanJet options program, which displays a message dialog containing an error message.

Also note that we call **exit()** if the call to **SCScanOpt()** succeeds. This is because the scanner options program appears to the user to be a dialog box associated with the scanning application that executed it. The ScanJet options program also provides an *Apply* button for changing scanner-specific settings without dismissing the scanner-specific options program.

Installation and Testing

After you have written your scanner driver and scanner options program, copy your scanner driver to the directory */usr/lib/scan/drv*, and the options program to the directory */usr/lib/scan/opt*. The driver and options program must have the same base name in order for *scanners*, the scanner installation tool, to recognize that they go together.

Next, run *scanners* to install your scanner. If it was already installed before you copied your options program to */usr/lib/scan/opt*, you must delete the scanner first (using the "Delete..." command on the Scanner menu).

Now you can run *gscan* to test your driver and options program. The "Scanner Specific Options..." command on the Parameters menu should bring up your options program.

Chapter 8

Generic Scanner Interface

This chapter describes the interface between a scanner driver and an
application program. Applications that use this interface can access any
scanner that conforms to the Impressario interface.

Generic Scanner Interface

This chapter describes the interface between a scanner driver and an application program.

The following major topics are discussed in this chapter:

- "Coordinate System for Scanning" on page 106
- "Data Structures" on page 107
- "Data Type Conventions" on page 108
- "Functions" on page 110

Overview

The generic interface between a scanner driver and an application program is flexible enough to accommodate a wide range of scanners and application programs. Application programs that use this interface can use any scanner that has a driver that supports this interface. Providing a driver for a particular scanner allows it to be accessed by any program written to use this interface.

The interface is implemented by a run-time library and a driver program. All drivers must provide the entry points necessary for the run-time library to provide the interface described in this document. In order to provide access to scanner-specific capabilities, scanner drivers are free to expand the interface, which is then accessed by scanner-specific programs that have standard ways of being invoked by application programs. (See Chapter 7, "Scanner-Specific Options.")

For more details on the functions described in this chapter, see the online manual pages.

Coordinate System for Scanning

When describing an area to be scanned, a coordinate system with the origin (0,0) in the upper left corner of the scannable area is used. The x-coordinate increases from left to right, and the y-coordinate increases from top to bottom.

Note that this is the upper left corner of the document being scanned; in the case of a flatbed scanner where the document is placed face down in the bed, this is the upper right corner or lower left corner of the bed.

When functions in the interface specify measurements along the horizontal and vertical axes of the scan area, the following units can be used to specify distance:

SC_INCHES   Specify measurements in inches.

SC_CENTIM   Specify measurements in centimeters.

SC_PIXELS   Specify measurements in pixels.

Data Structures

This section describes the generic scanner interface data structures.

SCANNER Data Structure

```
typedef struct tag_scanner {
        ...
} SCANNER;
```

The application maintains a pointer to a **SCANNER** data structure (obtained from **SCOpen()**) to specify the scanner to which operations should be applied.

SCDATATYPE Data Structure

Scanners support a variety of output data types. The **SCDATATYPE** structure encapsulates the common output data types produced by scanners.

```
typedef struct tag_scdatatype {
   unsigned int packing : 4;
   unsigned int channels : 4;
   unsigned int type : 8;
   unsigned int bpp : 8;
} SCDATATYPE;
```

The structure has these four fields:

packing   The *packing* field can take on the following values:

SC_PACKPIX   All the data for each pixel is stored together; for example, a line of 24-bit color is stored as RGBRGBRGB.

SC_PACKBAND   Each line of data is decomposed into its channels; for example, a line of 24-bit color is stored as RRRGGGBBB.

SC_PACKPLANE   All the data for a channel is stored separately from other channels. A page of color data is split into three pages of data; one for red, one for blue, and one for green.

channel	Number of components per pixel. Legal values are 1, 3, and 4. See *type* below.
type	Type of data. This parameter indicates how the data in the various channels is to be interpreted:

SC_MONO	Monochrome: 1 channel and 1 bit per pixel.
SC_GREY	Gray-scale: 1 channel.
SC_RGB	Red, green, and blue: 3 channels.
SC_CMY	Cyan, magenta, and yellow: 3 channels.
SC_CMYK	Cyan, magenta, yellow, and black: 4 channels.

bpp	Bits Per Pixel (per channel). The number of bits per pixel in each channel. For monochrome data, there is 1 bit per pixel. For 24-bit RGB color, there are 8 bits per pixel (x 3 channels = 24 bits).

Data Type Conventions

Generic scanner applications need not be written to support any particular data type. There are four basic data types that are typically used:

- Monochrome. *All* scanner drivers *must* support this monochrome format:
  - *packing* = SC_PACKPIX
  - *channels* = 1
  - *type* = SC_MONO
  - *bpp* = 1
- Eight-bit gray-scale. All scanner drivers that support any type of gray-scale or color output *must* support the following 8-bit gray-scale format:
  - *packing* = SC_PACKPIX
  - *channels*=1
  - *type* = SC_GREY
  - *bpp* = 8

- Planar 24-bit RGB color. The red, green, and blue channels are scanned in three separate passes; in this case, the data type format is
  - *packing* = SC_PACKPLANE
  - *channels* = 3
  - *type* = SC_RGB
  - *bpp* = 8
- Packed 24-bit RGB color. This applies to a one-pass color scanner that gets all of the data in one pass. The data type for color data from this kind of scanner is
  - *packing* = SC_PACKPIX
  - *channels* = 3
  - *type* = SC_RGB
  - *bpp* = 8

A scanning application that is prepared to deal with these four data types should be able to interact well with any well-behaved scanner driver.

Functions

Diagnostic Functions

Many of the functions specified here return 0 upon success and -1 in the event of a failure. If a function's return value indicates failure, the reason for the failure can be determined by examining the value of the global variable *SCerrno*. *SCerrno* will be between 0 and LASTERRNO (defined in */usr/include/sys/errno.h*) if the failure was due to a failed system call, and between SCEBASE and SCELAST (defined in *scanner.h*) if the failure was for some other reason. **#define** entries for the values between SCEBASE and SCELAST can be found in */usr/include/scanner.h*.

Table 8-1 lists the diagnostic functions.

Table 8-1   Diagnostic Functions

Function	Description
SCPerror()	Prints an error string.
SCErrorString()	Returns a character string containing an error message.

SCPerror() Function

```
void SCPerror(char *ident)
```

This function prints the value of *ident*, a colon, and a string of text corresponding to the current value of *SCerrno*.

SCErrorString() Function

```
char *SCErrorString(int err)
```

This function returns a character string containing a useful message describing the error condition represented by *err*. If *err* is not in the range 0 to LASTERRNO or SCEBASE to SCELAST, **SCErrorString()** returns a text string containing the words "Error code *err*," where *err* is the value passed to **SCErrorString()**.

Application/Driver Rendezvous Functions

Users refer to scanners by names given to them at install time. The installer uses scanners(1M), which adds entries to a mapping from scanner names to (*driver, device, options*) tuples. The mapping is contained in the file */var/scan/scanners*. The *driver* and *device* components are used to start the right driver on the device to access the scanner given by *name*, and *options* is the scanner-specific options program. *scanners* allows the specification of a default scanner.

The application/driver Rendezvous functions are listed in Table 8-2 and described below.

Table 8-2   Application/Driver Rendezvous Functions

Function	Description
SCOpen()	Prepares to perform operations on the scanner named by *name*.
SCOpenScreen()	Calls the screen scanner driver to scan from the specified screen.
SCOpenFile()	Calls the file scanner driver to scan from the specified file.
SCClose()	Breaks the connection between the application and the driver.
SCSetScanEnt()	Opens the scanner configuration file and returns a pointer.
SCGetScanEnt()	Gets a **SCANENT** structure for each installed scanner.
SCEndScanEnt()	Frees the resources used to enumerate scanners.
SCScannerName()	Returns the name associated with the scanner at installation.
SCScannerEnt()	Returns the **SCANENT** structure of an open scanner.
SCDefaultScannerName()	Gets the default scanner name, if any.

SCOpen() Function

```
SCANNER *SCOpen(char *name)
```

The **SCOpen()** function prepares to perform operations on the scanner named by *name* by starting the appropriate driver on the appropriate device. **SCOpen()** performs the lookup in the *name* -> (*driver*, *device*, *options*) mapping. If *name* is NULL, the default scanner is used.

SCOpen() returns a pointer to a **SCANNER** struct if successful, or NULL if there is an error. If *name* is NULL and no default scanner has been set, **SCOpen()** opens the first scanner found in */usr/lib/scan/scanners*.

SCOpenScreen() Function

```
SCANNER *SCOpenScreen(char *screen)
```

This function invokes the screen scanner driver to scan from the specified screen. It returns a pointer to a **SCANNER** struct if successful, NULL if there is an error.

SCOpenFile() Function

```
SCANNER *SCOpenFile(char *file)
```

This function invokes the file scanner driver to scan from the specified file. It returns a pointer to a **SCANNER** struct if successful, NULL if there is an error.

SCClose() Function

```
int SCClose(SCANNER *s)
```

This function breaks the connection between the application and the driver program. It returns 0 on success, -1 if there is an error.

SCSetScanEnt() Function

```
FILE *SCSetScanEnt(void)
```

This function opens the scanner configuration file. It returns a pointer to the open **FILE** structure on success, NULL if there is an error.

SCGetScanEnt() Function

```
typedef struct tag_scanent {
   char *name;
   char *driver;
   char *device;
   char *options;
} SCANENT;

SCANENT *SCGetScanEnt(FILE *fp)
```

To get a **SCANENT** structure for each scanner installed on the system, this function should be called repeatedly until it returns NULL. The contents of the memory pointed to by the return value of **SCGetScanEnt()** are undefined after any subsequent calls to this function, so copy the return value if you need to preserve it across calls to **SCGetScanEnt()**.

SCEndScanEnt() Function

```
int SCEndScanEnt(FILE *fp)
```

This function frees the resources used to enumerate scanners. It returns 0 on success, -1 if there is an error.

SCScannerName() Function

```
char *SCScannerName(SCANNER *s)
```

This function returns the name associated with the scanner at installation. Applications can use this to get at the name of the default scanner being used if **SCOpen()** was called with NULL. It returns a pointer to a character string on success, NULL if there is an error. The memory pointed to by the return value of **SCScannerName()** belongs to *libscan* and should not be modified or freed.

SCScannerEnt() Function

```
SCANENT *SCScannerEnt(SCANNER *s)
```

This function returns a **SCANENT** structure describing an open scanner. The memory pointed to by the returned value of **SCScannerEnt()** belongs to *libscan* and should not be modified or freed.

SCDefaultScannerName() Function

```
char *SCDefaultScannerName(void)
```

This function gets the default scanner name, if any. It returns the name of the default scanner, or NULL if no default scanner has been set. The memory pointed to by the returned value of **SCDefaultScannerName()** belongs to *libscan* and should not be modified or freed.

Scanning Resolution Functions

Scanners typically support a range of resolutions (pixels per inch). Scanner drivers should support any resolution between the minimum and maximum resolutions supported by the scanner, decimating or replicating pixels as necessary to support the requested resolution. This gives the application the opportunity to preview the scanning area in an arbitrarily sized window.

It is not the scanner driver's responsibility to perform higher-quality scaling of the image data. **SCGetScannerRes()** can be used by the scanner application to determine which resolutions are supported directly by the scanner without decimation or replication by the driver.

SCGetScannerRes() Function

```
int SCGetScannerRes(SCANNER *s, int metric,
    float **xres, float **yres, int *nres)
```

This function returns arrays of hardware-supported resolutions. The *xres* and *yres* arrays specify supported horizontal and vertical resolutions. *metric* should be one of SC_INCHES or SC_CENTIM. *nres* sets the number of resolution pairs in the *xres* and *yres* arrays. **SCGetScannerRes()** returns 0 if successful, -1 if there is an error.

SCGetMinMaxRes() Function

```
int SCGetMinMaxRes(SCANNER *s, int metric,
    float *minx, float *miny, float *maxx, float *maxy);
```

This function determines the resolution bounds; that is, the minimum and maximum horizontal and vertical resolutions that the scanner supports. It is an error to call **SCGetMinMaxRes()** with *metric* equal to SC_PIXELS. **SCGetMinMaxRes()** returns 0 if successful, -1 if there is an error.

Scanning Area Functions

A scan may be limited by the application to a subset of the scannable area supported by the scanner. **SCGetPageSize()** is provided so that applications can determine the size of the scannable area supported by the scanner.

SCGetPageSize() Function

```
int SCGetPageSize(SCANNER *s, int metric, float *x, float *y,
                  float *width, float *height)
```

This function gets the entire scannable area. It is an error to call it with *metric* equal to SC_PIXELS. **SCGetPageSize()** returns 0 if successful, -1 if there is an error.

SCGetDataTypes() Function

```
int SCGetDataTypes(SCANNER *s, SCDATATYPE **dt, int *ntypes)
```

This function sets **dt* to point to an array of the data types supported by the scanner driver. *ntypes* gets the number of data types supported. The memory pointed to by **dt* belongs to *libscan* and should not be modified or freed. It should also not be expected to retain its values after subsequent calls to **SCGetDataTypes()**.

Scanning Functions

After **SCOpen()** has been called, the scanner is idle. In order to initiate a scan, the functions **SCSetup()** and **SCScan()** are called. Characteristics of the data to be scanned can be determined with **SCGetScanSize()**. A scan in progress can be aborted at any time with the function **SCAbort()**. The scanner status can be determined by calling the function **SCGetStatus()**.

Table 8-3 lists the available scanning functions.

Table 8-3   Scanning Functions

Function	Description
SCSetup()	Prepares the scanner for a scan.
SCGetScanSize()	Determines the width, height, and number of bytes per scan line.
SCScan()	Starts scanning.
SCGetScanLine()	Retrieves scan line data.
SCDataReady()	Determines whether any data is available.
SCGetFD()	Returns the file descriptor for scan data.
SCScanFD()	Starts scanning (alternative call).
SCAbort()	Aborts the current scan.
SCGetStatus()	Gets the status of the scanner.
SCGetStatusFD()	Returns a file descriptor for scan status.

SCSetup() Function

```
SCSetup(SCANNER *s, int preview, SCDATATYPE *type,
      int rmetric, float xres, float yres,
      int wmetric, float x, float y, float width,
      float height)
```

This function is used to prepare the scanner for a scan. The type of data, the resolution, and the scanning area are specified. *preview* is nonzero if this is a "preview" scan; that is, when the driver is faced with a trade-off between speed and image quality, it should choose speed, because this is not the "real" scan. After calling **SCSetup()**, **SCScan()** is called to initiate scanning. **SCSetup()** returns 0 if successful, -1 if there is an error.

SCGetScanSize() Function

```
int SCGetScanSize(SCANNER *s, int *width, int *height,
                  int *bytesPerLine)
```

This function is called after **SCSetup()** to determine the width, height, and number of bytes per scan line that will be returned by the driver. It returns 0 if successful, -1 if there is an error.

SCScan() Function

```
int SCScan(SCANNER *s)
```

This function tells the driver to start scanning. The driver immediately starts to scan and buffer the data. **SCScan()** does not fetch any scan data (see **SCGetScanLine()**). **SCScan()** returns 0 if successful, -1 if there is an error.

SCGetScanLine() Function

```
int SCGetScanLine(SCANNER *s, void *buf, int bytes)
```

This function retrieves scan line data. *bytes* should be set to the number of bytes in a scan line as determined by **SCGetScanSize()**.

Note that for color planar data, **SCGetScanLine()** is called once for each line in each plane of data. For 100 lines of 24-bit RGB planar data, **SCGetScanLine()** is called a total of 300 times, with the first 100 calls retrieving the red plane, the second 100 calls retrieving the green plane, and the third 100 calls retrieving the blue plane.

SCDataReady() Function

```
int SCDataReady(SCANNER *s)
```

This function is used to determine whether any data is available for calls to **SCGetScanLine()**; that is, whether a call to **SCGetScanLine()** will block waiting for data to become available.

SCDataReady() returns 1 if data is available (**SCGetScanLine()** will not block), 0 if no data is available (**SCGetScanLine()** will block), or -1 if there is an error. It is an error to call **SCDataReady()** if scanning was started by a call to **SCScanFD()**.

SCGetFD() Function

```
int SCGetFD(SCANNER *s)
```

This function returns the file descriptor over which scan data from the scanner driver comes. Checking the state of this descriptor with the select(2) system call is equivalent to calling **SCDataReady()**. If **SCScanFD()** was called, **SCGetFD()** returns the file descriptor that was passed to that function. **SCGetFD()** returns -1 if there is an error.

SCScanFD() Function

```
int SCScanFD(SCANNER *s, int fd)
```

This function is an alternative to calling **SCScan()** to start scanning and **SCGetLine()** to fetch the data. After **SCScanFD()** is called, the driver writes the scanned data to *fd*; this is useful if the output data format of the scanner interface matches the input data type of another interface. **SCScanFD()** returns 0 if successful, -1 if there is an error.

SCAbort() Function

```
int SCAbort(SCANNER *s)
```

This function aborts the current scan. Data buffered by the driver is discarded. **SCAbort()** returns 0 if successful, -1 if there is an error.

SCGetStatus() Function

```
enum scstate { SC_READY, SC_SCANNING, SC_ERROR };
typdef struct tag_scstatus {
    enum scstate state;       /* ready, scanning, error */
    int errno;                /* only valid if state == SC_ERROR */
    long curline;             /* current line being scanned */
    int pass;                 /* current scanning pass */
} SCSTATUS;
```

```
int SCGetStatus(SCANNER *s, SCSTATUS *st)
```

This function gets the status of the scanner. It returns 0 if successful, -1 if there is an error.

SCGetStatusFD() Function

```
int SCGetStatusFD(SCANNER *s)
```

118

This function returns a file descriptor that can be passed to the select(2) system call. When *select* indicates that the file descriptor is ready for reading, the scanner driver has updated the scanning status. Retrieve the status by calling **SCGetStatus()**; do NOT pass the file descriptor returned from **SCGetStatusFD()** to any other system call.

SCGetStatusFD() provides a mechanism whereby it is not necessary for an application to periodically call **SCGetStatus()** in a timer loop to detect changes in scanner status. **SCGetStatusFD()** returns -1 if there is an error.

Document Feeder Functions

The scanner interface has provisions for the support of scanners that have document feeders attached. This facilitates the development of applications that can scan multiple pages without user intervention. Table 8-4 lists the document feeder functions.

Table 8-4   Document Feeder Functions

Function	Description
SCFeederGetFlags()	Gets the feeder flags.
SCFeederSetFlags()	Sets feeder flags.
SCFeederAdvance()	Advances the feeder to the next document.
SCFeederReady()	Checks if the feeder is ready for feeding.

SCFeederGetFlags() Function

```
typedef unsigned int SCFEEDERFLAGS;

#define SC_HASFEEDER 1
#define SC_AUTOFEED  2
#define SC_PROGFEED  4

int SCFeederGetFlags(SCANNER *s, SCFEEDERFLAGS *flags);
```

This function fills in the flags variable with flags appropriate for the scanner associated with *s*. If SC_AUTOFEED and SC_PROGFEED are both set, **SCFeederSetFlags()** should be called before any calls to **SCScan()** to establish how the application is to interact with the feeder. The meanings of the flags are as follows:

SC_HASFEEDER   Set if there is a document feeder attached to the scanner.

SC_AUTOFEED   Set if the feeder can operate such that each call to
                       SCScan() causes the next document to be loaded.

SC_PROGFEED   Set if the feeder can operate so that SCScan() can be
                       called multiple times per document. It is necessary to
                       call SCFeederAdvance() to load the next document.

SCFeederGetFlags() returns 0 if successful, -1 if there is an error.

SCFeederSetFlags() Function

```
int SCFeederSetFlags(SCANNER *s, SCFEEDERFLAGS flags);
```

This function should be called before calling **SCScan()** for scanners in which **SCFeederGetFlags()** sets both SC_AUTOFEED and SC_PROGFEED. After calling **SCFeederSetFlags()**(*s*, SC_AUTOFEED), the feeder advances to the next document after every call to **SCScan()**. After calling **SCFeederSetFlags()**(*s*, SC_PROGFEED), a call to **SCFeederAdvance()** is necessary to advance to the next document. **SCFeederSetFlags()** returns 0 if successful, -1 if there is an error.

SCFeederAdvance() Function

```
int SCFeederAdvance(SCANNER *s)
```

This function advances the feeder to the next document. This call is valid only if the scanner supports the SC_PROGFEED mode. For scanners that support both the SC_AUTOFEED and SC_PROGFEED modes, **SCFeederSetFlags()**(*s*, SC_PROGFEED) must have been called previously.

SCFeederAdvance() returns 0 if successful, -1 if there is an error. When unloading the last document, this function returns -1 with *SCerrno* set to SCFEEDEREMPTY.

SCFeederReady() Function

```
int SCFeederReady(SCANNER *s)
```

This function checks if the feeder is ready for feeding. **SCFeederReady()** returns 0 if the feeder is ready, -1 if not. If the feeder is empty, *SCerrno* is set to SCFEEDEREMPTY; if any other error conditions exist, *SCerrno* is set appropriately.

Note that this function needs to be called before **SCFeederAdvance()** to determine whether a document is ready to be scanned after the call to **SCFeederAdvance()**.

Events

Scanner applications need to be aware that the configuration information about a scanner obtained from **SCGetMinMaxRes()**, **SCGetScannerRes()**, **SCGetPageSize()**, and **SCGetDataTypes()** can change. This typically happens when the user selects a new input medium using the scanner specific options panel. For example, some scanners support transparency options that have a different scanning page size than the normal scanning bed. When the user decides to scan transparencies, the driver notifies the application that it needs to call **SCGetPageSize()** by sending an event. See Table 8-5.

Table 8-5   Event Functions

Function	Description
SCGetEvent()	Receives an event from the scanner driver.
SCEventPending()	Tests whether an event is currently pending.
SCGetEventFD()	Obtains an event file descriptor for passing to *select*.

SCGetEvent() Function

```
typedef struct tag_infoChange {
    unsigned int pageSizeChanged : 1;
    unsigned int resolutionChanged : 1;
    unsigned int dataTypesChanged : 1;
    unsigned int feederFlagsChanged : 1;
} SCINFOCHANGE;

#define SCEVENT_INFOCHANGE 1

typedef struct tag_scevent {
    unsigned int eventType;
    union {
        SCINFOCHANGE infoChange;
    } event;
} SCEVENT;

int SCGetEvent(SCANNER *s, SCEVENT *event)
```

SCGetEvent() is called to receive an event from the scanner driver. The event structure should be examined, and if the *pageSizeChanged* field is set, the application should call **SCGetPageSize()** to query the new page size; if the *resolutionChanged* field is set, the application should call **SCGetMinMaxRes()** and **SCGetScannerRes()** to query the new resolutions; if the *dataTypesChanged* field is set, the application should call **SCGetDataTypes()** to query the new data types, and if the *feederFlagsChanged* field is set, the application should call **SCGetFeederFlags()** to query the new feeder flags.

SCEventPending() Function

```
int SCEventPending(SCANNER *s)
```

SCEventPending() is called to test whether or not an event is currently pending. If an event is pending, the application should call **SCGetEvent()** to receive it. **SCEventPending()** returns 1 if any events are pending, and 0 if no events are pending.

SCGetEventFD() Function

```
int SCGetEventFD(SCANNER *s)
```

SCGetEventFD() returns a file descriptor that can be passed to the *select* system call. When *select* indicates that this file descriptor is ready for reading, then an event is pending and the application should call **SCGetEvent()** to retrieve it.

122

Testing for Impressario Compatibility

This chapter explains how to use the programs included in Impressario to test the output capabilities of an Impressario-supported printer, and test an Impressario printer software installation.

Testing for Impressario Compatibility

This chapter explains how to use the programs that test printing compatibility with the Impressario environment.

The following topics are discussed in this chapter:

Testing Impressario Printing Compatibility

The Impressario Developer's Kit provides two programs for testing printing compatibility with the Impressario environment. The *testipr* program tests the output capabilities of an Impressario supported printer. The *testiconfig* program tests an Impressario printer software installation. Both of these programs are located in the directory */usr/impressario/tests/print.*The use of these test harnesses is not sufficient testing to ensure the quality of your product. They are only helpful tools, not a substitute for additional testing.

Testing an Impressario Printer

A printer supported by Impressario can print a wide range of file formats with a large selection of printing options for each file format. Testing each supported file format and printing option can be a laborious task if done manually, one test case at a time. *testipr* automates the testing process by printing a set of standard test files, according to a standard test plan.

testipr is in the directory */usr/impressario/tests/print* and typically is run from there. The name of a printer installed on the system is the only required command-line parameter. It is recommended that the printer be physically connected to the system on which *testipr* is run. It is the responsibility of the user to ensure that the printer is not used by other users during testing.

On startup, *testipr* looks in the */usr/impressario/tests/print* directory for test configuration files. These files are identified by a *.ipr* suffix and a basename corresponding to the test class name. There are three standard configuration files: *text.ipr*, *image.ipr*, and *postscript.ipr*. These files contain the text, image, and PostScript test classes, respectively. Each configuration file describes a set of tests to be run by *testipr*. These files should not be modified. If you wish to create your own test cases, copy the existing configuration files to a new location, modify them to suit your needs, and use the **-p** command-line option to tell *testipr* where to find the files.

The format of the configuration file follows that of a X Window System resource file. For example:

```
! This is a comment in an example .ipr file

test.basePath: /usr/impressario/data

test.1.file: testfile.sgi
test.1.options: -rotate 90 -gamma 3.5
test.1.desc: "SGI Image File - rotated 90, gamma 3.5"

test.2.file: testfile.sgi
test.2.options: -rotate 90 -gamma 3.5
test.2.desc: "SGI Image File - rotated 90, gamma 3.5"
```

Each resource must start with the keyword **test**. *basePath* is an optional resource that specifies a directory path. If it is specified, the path will be prepended to each file resource. The *file* resource specifies a test file to print. Typically, these test files are from */usr/impressario/test/data*. The *options* resource specifies the actual -**o** model file options to be used in the test. Note that the -**o** should not be specified. The *testipr* program automatically prepends the **nobanner** option to all option strings. The option string is passed as the argument to the *lp* command's -**o** option. The *desc* resource provides a description of the test. This string is written to the test log file. Each test case must be numbered consecutively, starting with 1.

The file specified in each test is submitted for printing using the specified printing options. A log file, called */var/tmp/testipr.<printerName>.log*, is created. This log file contains general information about the printer being tested and its host environment. The log also contains a detailed list of all tests performed and their corresponding spooling system print job IDs. A complete test record consists of the printer output, the corresponding log file, and the */var/spool/lp/log* file. See the testipr(1) reference page for command-line options and the most up-to-date information on this test program.

Example 1:

Run all tests on the printer hp4.

testipr hp4

Example 2:

Run only image tests on the printer hp4.

```
testipr -c image hp4
```

Example 3:

Run only image tests numbers 5 and 6 on the printer hp4.

```
testipr -c image -t 5,6 hp4
```

Testing an Impressario Printer Software Installation

An Impressario supported printer has greatly enhanced printing capabilities over other types. To provide these enhanced capabilities, software complying with Impressario specifications must be installed in standard locations. The *testiconfig* program checks that the software support for a printer conforms to Impressario specifications. Typical users of this program are third-party printer developers who wish to verify that their printer support is compatible with the Impressario printing environment.

testiconfig performs a number of checks to ensure conformance to Impressario printer support specifications. The program performs checks on the printer model file, POD files, graphical options panel and printer driver. All output is sent to the standard output. If the **-v** option is specified, additional information is displayed during the test. The *testiconfig* program requires the name of the printer model file. The printer support software must be installed on the system on which the *testiconfig* is run. Note that a printer need not be physically installed on the system or installed by the spooling system to run this test program.

For example:

Test the installation for an HP LaserJet 4:

```
testiconfig -v laserjetPJL_model
```

See the testiconfig(1) reference page for command-line options and the most up-to-date information on this program.

Packaging Your Impressario Product

This chapter explains how to package your Impressario product to make distribution simple and consistent with the Impressario printing and scanning environment.

Packaging Your Impressario Product

This chapter explains how to package your Impressario Product.

The following topics are discussed in this chapter:

- "Making a tar Archive for Software Distribution" on page 132
- "Packaging Impressario Printing Software" on page 133
- "Packaging Impressario Scanning Software" on page 137

Overview

Impressario provides an open printing and scanning environment. Third-party support for printers and scanners can be added to the Impressario environment by following the procedures described in this chapter.

There are two methods of packaging Impressario software for distribution: the *swmgr* software packaging and installation technology and *tar* archives. We recommend using *swmgr*, which users access with the Software Manager option in the System Toolchest, because of its flexibility and ease of use. Instructions for creating an image that can be installed by *swmgr* are in the *Software Packager User's Guide*. In the next section, Table 10-1 (for printers) and Table 10-2 (for scanners) define the locations, ownership, and privileges of the files that are typically needed to create an Impressario product.

If you prefer to create a *tar* archive, the following section describes how to create one for installing Impressario drivers.

Making a tar Archive for Software Distribution

To create a *tar* archive for your software distribution, use the following procedure:

1. Become superuser. The creation of all tape archives and the subsequent installation of the product by the end user must be done as superuser. Becoming superuser is typically accomplished by either logging in as *root* or executing the *su* command. Typically, a password must be provided to gain superuser access to a system. Ask your system administrator for assistance.

2. Copy the files that compose your product from your development area to the directories into which they will be installed. See "Packaging Impressario Printing Software" on page 133 and "Packaging Impressario Scanning Software" on page 137 for the files typically installed by printing and scanning products.

3. Change the permissions and ownership of each file according to the recommendations in "Packaging Impressario Printing Software" and "Packaging Impressario Scanning Software." The *chmod* command is used to change file permissions and the *chown* command is used to change file ownership. For example, to give the file *foo* read–write permission for the owner and read-only permission for all others, and to specify a *root* owner and *sys* group, enter the following commands:

    ```
    chmod 0644 foo
    chown root.sys foo
    ```

4. Enter the *tar* command and specify the *absolute* pathname of each file that is part of the distribution. For example, to create an archive consisting of two files and to place that archive on the default tape device, enter the following command:

    ```
    tar cvLf /dev/tape /usr/lib/print/mydriver /var/spool/lp/model/mymodel
    ```

    For detailed information on the *tar* command, refer to the tar(1) manual page.

5. Optionally, you may include in your distribution a shell script that removes the files that are installed by your product. This allows customers to reclaim the disk space used by your product if your product is no longer being used.

Packaging Impressario Printing Software

To illustrate better the process of packaging Impressario printing support software, let us create a fictitious product. The product provides Impressario support for the Blast family of printers. The Blast product line consists of the Blast 1, Blast 2CVX, and the Blast P+. Because each of the Blast printers provides similar functions, support for all of them is provided by a single model file.

Impressario printing support products typically are named with the printer family, the word *Print*, and the lowest version number of Impressario that supports the product. In keeping with this convention, the product is named "BlastPrint for Impressario 2.0."

A typical Impressario printer-support product consists of the following files:

Model file   When the printer is registered with the spooling system, this file is copied and becomes the printer interface file. The spooling system executes this shell script for each print job.

Printer driver   This executable program communicates with the printer. The interface file invokes the driver to do the actual printing of a file. The driver sends data to the printer and updates the POD status file.

POD files   The POD consists of three files, all with the same base name as the model file but with the suffixes *.config*, *.status*, and *.log*. These plain text files contain static and dynamic printer information.

Graphical options panel program
                 This GUI program provides graphical access to printer-specific options for users. The program is given the same base name as the model file with the suffix *.gui*.

Graphical options panel resource file
                 This is an X Window system resource file for the graphical options panel program. The file is named with the class name of the graphical options panel program. This name must also match the *GUI_CLASS* variable in the model file.

Manual page   A manual page that describes the printing product should be included. By convention, this manual page is named with the product name. The manual page must be formatted using *nroff* and must be compressed before installation. A product reference page template and a Makefile to perform the required formatting, compression, and installation are provided in the directory */usr/impressario/man*.

                 Note: In order to create a reference page you must have the Documenter's Workbench product installed on the development system.

The files listed in Table 10-1 comprise the BlastPrint product. The files are listed with their absolute pathnames, permissions, and ownership.

Table 10-1 Typical Printing Product Files

Description	Pathname	Permissions	Owner
Model file	/var/spool/lp/model/blast_model	0755	lp.lp
Printer driver	/usr/lib/print/blaster	0755	lp.lp
POD files	/usr/lib/print/data/blast_model.config	0664	lp.lp
	/usr/lib/print/data/blast_model.status	0664	lp.lp
	/usr/lib/print/data/blast_model.log	0664	lp.lp
Graphical options program	/var/spool/lp/gui_model/ELF/blast_model.gui	0755	lp.lp
Graphical options resources	/usr/lib/X11/app-defaults/Blast	0644	root.sys
Reference page	/usr/share/catman/u_man/cat1/blastprint.z	0444	root.sys

Before creating the actual software distribution, the above files must be copied to the directories indicated and given the specified ownership and permissions. Once this is done, the Impressario test program testiconfig(1) can be run to verify that the product conforms to Impressario product installation specifications.

To run the *testiconfig* command on the BlastPrint product, execute the following:

```
cd /usr/impressario/tests/print
./testiconfig blast_model
```

Once the product installation has been verified, a software distribution can be created. This is done using the *tar* command. Assuming you are making a tape distribution, you would issue the following command (note the use of the line continuation character "\" to allow the command line to extend over multiple lines):

```
tar cvLf /dev/tape /var/spool/lp/model/blast_model \
              /usr/lib/print/blaster \
              /usr/lib/print/data/blast_model.config \
              /usr/lib/print/data/blast_model.status \
              /usr/lib/print/data/blast_model.log \
              /var/spool/lp/gui_model/ELF/blast_model.gui \
              /usr/lib/X11/app-defaults/Blast \
              /usr/share/catman/u_man/cat1/blastprint.z
```

The resulting tape archive represents the BlastPrint product.

It is recommended that copying the files to their installation directories, assigning ownership and permissions to the files, and archiving the files be automated in a shell script. This eliminates a lot of typing and provides a consistent distribution mechanism.

Once the distribution tape has been created, it should be installed on a new system, that is, one that has never had BlastPrint. This is done as superuser using the command

```
tar xvf /dev/tape
```

Once BlastPrint has been installed, the *testiconfig* program should again be run to verify that the installation is complete and correct. A printer should then be connected to the system and registered with the spooling system using the Printer Manager (printers). The Impressario test command *testipr* should be used with the newly installed printer to verify that it is able to print all supported Impressario file formats and options.

If the printer was installed with the name **myblaster**, it can be tested by executing the commands

```
cd /usr/impressario/tests/print
./testipr myblaster
```

This completes the creation of an Impressario printer-support product.

Packaging Impressario Scanning Software

To illustrate the process of packaging Impressario scanning software, you will create a fictitious product to provide Impressario support for the LowTech 100 scanner.

Impressario scanning products are typically named with the scanner family, the word *Scan*, and the lowest version number of Impressario that supports this product. Name your product "LowTechScan for Impressario 2.0."

A typical Impressario scanner-support product consists of the following files:

Scanner driver   The executable program that obtains the data from a scanner.

Graphical options program
> This is the graphical scanner-specific options program launched from a scanning application. The program showcases a scanner's features. This program must have the same base name as the scanner driver in order to be recognized by scanners, the scanner installation tool.

Graphical options resource file
> This is the X Window System resource file for the graphical options program and is named with the class name of the graphical options program. The class name must be the same as the graphical options program name, with the first letter capitalized. If this resource file is not named using these conventions, the options program will not have access to its resources when invoked for a network scanner.

Reference page   A reference page should be created that describes the scanning product. By convention, this reference page is named with the product name. The reference page must be formatted using *nroff* and must be compressed before installation. A product reference page template and a Makefile to perform the required formatting, compression, and installation are provided in the directory */usr/impressario/man*.

> **Note:** To create a reference page, you must have the Documenter's Workbench product installed on the development system.

The files listed in Table 10-2 compose the LowTechScan product. The files are listed with their absolute pathnames, permissions, and ownership.

Table 10-2   Typical Scanning Product Files

Description	Pathname	Permissions	Owner
Scanner driver	/usr/lib/scan/drv/lowtech	0755	root.sys
Graphical options program	/usr/lib/scan/opt/lowtech	0755	root.sys
Graphical options resources	/usr/lib/X11/app-defaults/Lowtech	0644	root.sys
Manual page	/usr/share/catman/u_man/cat1/lowtech.z	0444	root.sys

Before creating the actual software distribution, copy the above files to the directories indicated and give them the specified ownership and permissions. Then run the scanners tool to verify that the correct string for the LowTech 100 scanner appears in the "Install New Scanner" dialog. Finally, install a LowTech 100 scanner using the *scanners* tool, and make sure that *gscan* is able to run the graphical options program from the "Scanner Specific Options" command in the Parameters menu.

Once the product installation has been verified, a software distribution can be created with the *tar* command. Assuming you are making a tape distribution, you would issue the following command (note the use of the line continuation character "\" to allow the command line to extend over multiple lines):

```
tar cvLf /dev/tape /usr/lib/scan/drv/lowtech \
        /usr/lib/scan/opt/lowtech \
        /usr/lib/X11/app-defaults/Lowtech \
        /usr/share/catman/u_man/cat1/lowtech.z
```

The resulting tape archive represents the LowTechScan product.

It is recommended that copying the files to their installation directories, assigning ownership and permissions to the files, and archiving the files be automated in a shell script. This eliminates a lot of typing and provides a consistent distribution mechanism.

Once the distribution tape has been created, it should be taken to a new system (in other words, one that has never had LowTechScan installed) and installed. This is done using the following command as superuser:

```
tar xvf /dev/tape
```

Once LowTechScan has been installed, the *scanners* and *gscan* programs should again be run to verify that the installation is complete and correct.

This completes the creation of an Impressario scanner-support product.

Enhancing Impressario with Plug-Ins

This chapter explains how to add new features to the Impressario open architecture. It describes how Impressario's automatic filetype recognition and conversion pipeline works, how to add new file types, and how to add a new PostScript Raster Image Processor (RIP).

Enhancing Impressario With Plug-Ins

This chapter explains how you can add plug-ins to Impressario to satisfy needs that haven't been integrated into the base software. This allows you to dynamically update the Impressario feature set.

The following topics are discussed in this chapter:

- Impressario automatic file type recognition and file conversion pipeline

- adding new file types to those that Impressario already recognizes and prints

- adding a new PostScript Raster Image Processor (RIP) to Impressario and switching between the standard Impressario RIP and your new RIP

How the Impressario File Conversion Pipeline Works

The key components of the Impressario file conversion pipeline are

- a database of file type rules (FTRs)

- the *wstype* runtime filetype recognition utility

- the *fileconvert* file conversion utility

File Type Rules

The file type rules of the FTR database are documented in the *Indigo Magic Desktop Integration Guide*, which is available as an online book and can be installed from your IRIX CD.

The file type rules have two parts relevant to printing. The first is a TYPE rule for using the first 512 bytes of a file to recognize that the file is a particular type. The second is a CONVERT rule for converting that type to another type; the ultimate goal is to be able to convert any file type to a set of printable file types. Each CONVERT rule contains a pair of known types (a source and a destination), a Bourne shell command for converting from the source to the destination, and the computational cost of the conversion. Impressario builds on the standard IRIX database of filetypes by extending the FTR set to handle additional input file types, and by adding new destination file types.

Runtime File Type Recognition Utility

The *wstype* utility is used to determine the file type of a file or set of files. It is similar to file(1) in basic operation, but has no hardcoded special cases and no */etc/magic* file, relying on the compiled FTR database for typing information.

See the wstype(1) reference page for additional information.

File Conversion Utility

The *fileconvert* utility builds a directed acyclic graph out of all known file types, determines the input file type, and attempts to find the lowest-cost path from source to destination. The cost analysis is necessary because the file conversion is usually a multiple-step process.

For example, suppose the goal is to print a GIF on a raster printer. There is no filter to directly convert a GIF to raster data, so *fileconvert* might generate a couple of alternate paths. One path is shown below:

Conversion	Filter
GIF to TIFF	/usr/lib/print/il2stiff
TIFF to PostScript	/usr/lib/print/stiff2ps
PostScript to Raster	/usr/lib/print/psrip

A second path is shown below:

Conversion	Filter
GIF to TIFF	/usr/lib/print/il2stiff
TIFF to SGI	/usr/lib/print/tiff2sgi
SGI to PostScript	/usr/lib/print/sgi2ps
PostScript to Raster	/usr/lib/print/psrip

Each of the filters has an associated computational cost. *fileconvert* determines the total cost of each path and chooses the lowest-cost route.

If a conversion path exists, *fileconvert* prints a Bourne shell command string which, when executed, generates the destination file type on standard out. Most model files simply execute this shell command string, piping the results to the printer driver. This flexible, extensible mechanism enables Impressario to print virtually any recognizable file type on any printer that is compliant with Impressario.

See the fileconvert(1) reference page for additional information.

Adding a New Filetype to Impressario

To extend Impressario to make a new type of file printable, you need only perform the following steps:

1. Write a filter to convert your new type.

2. Write an FTR rule that recognizes the new filetype.

3. Add a CONVERT rule from that type to a known Impressario type.

4. Install and test your changes.

Writing a New Filter

First, write a filter to convert your new type to either STIFF or PostScript, the two main formats for input to Impressario. The *fileconvert* utility requires that your filter be able to accept input either on standard in or from a specified filename. It should also be able to write output to standard out without using intermediate files, so that the conversion can succeed, even if the converting host has low disk space. See the *Indigo Magic Desktop Integration Guide*, Chapter 5, for more details. See the gif2stiff(1) reference page for an example.

Writing an FTR

Next, write an FTR that can recognize the new filetype given only the first 512 bytes of the file. See the *Indigo Magic Desktop Integration Guide* and the ftr(1) reference page for a description of the available primitives for matching a file's type and for a description of the additional parts of a good FTR.

Adding a CONVERT Rule

As part of your FTR, add a CONVERT rule from that type to a known Impressario type. We strongly recommend that you convert image types to the *STIFF93aImage* type and convert other non-image file formats to the *PostScriptFile* type. Refer to Appendix A and the libstiff(3) reference page for information on writing STIFF output. Note that once you convert to either a *PostScriptFile* or a *STIFF93aImage*, Impressario knows how to convert that filetype to a printable format on any printer.

Model files that are compliant with Impressario also automatically set the proper environment variables for the options the user selected on the spooler command line, and the standard Impressario CONVERT rules automatically pick up the appropriate environment variables when converting these file types to printable types. This is how user-specified printing options are applied to the dynamically-constructed file conversion pipeline.

Installation and Testing

The easiest way to show how to test a new filetype is by example.

Setting Up an Example

The example makes the following assumptions:

- The new filetype is called *BlastFile.*
- The new FTR is in the directory */usr/tmp/blastfile.ftr.*
- A sample BlastFile is in the directory */usr/tmp/sample.blastfile.*
- All BlastFiles start with the word *blastfile.*

A sample (although incomplete) TYPE rule for the example would look like this:

```
TYPE BlastFile
    MATCH string(0,9) == "blastfile";
    LEGEND Blast Image File

CONVERT BlastFile STIFF93aImage
    COST 50
    FILTER /usr/lib/print/blast2stiff $IMPR_IMG2STIFFOPTS
```

Testing the New Filetype

1. To test the new filetype, become root by entering:

   su

   Supplying a password, if necessary.

2. Copy your FTR into */usr/lib/filetype/install*:

   cp /usr/tmp/blastfiletype.ftr /usr/lib/filetype/install

3. Compile the FTR database:

   cd /usr/lib/filetype ; make

   See the ftr(1) reference page if any errors occur.

4. Run *wstype* to verify that Impressario now recognizes your file type:

   wstype /usr/tmp/sample.blastfile

   You should see the following output:

   ```
   /usr/tmp/sample.blastfile BlastFile
   ```

5. Try to convert that file type to a printable file type. The most printable file type in the Impressario database is ImpressarioPostScriptFile, so try that:

   **fileconvert -d ImpressarioPostScriptFile \
   /usr/tmp/sample.blastfile**

   You should get back something like:

   ```
   PRINTFILES="/usr/tmp/sample.blastfile" ; /usr/lib/print/blast2stiff
   $IMPR_IMG2STIFFOPTS $PRINTFILES | /usr/lib/print/stiff2ps
   $IMPR_SGI2PSOPTS
   ```

   Note: The above output appears on one line.

   If *fileconvert* does not return a valid shell command string, check the exit code of the filter. If you are in the C shell, enter:

   echo $status

   If you are in the Bourne shell, enter:

   echo $?

   If the exit code is not zero, then *fileconvert* was unable to convert your filetype. Verify that your destination filetype is convertible to a printable type by checking the CONVERT rules in the FTR database and making sure there is a conversion path from your destination type to the Impressario type, then try the above steps again.

6. Once steps 1-5 work, verify that your filter produces valid output by entering

   vstiff /usr/tmp/sample.blastfile

   This should bring up the *vstiff* previewer, which uses the FTR database of TYPE and CONVERT rules to convert to a screen-viewable STIFF file. Use the "PostScript Options…" menu choice on the File menu to choose different color spaces and depths. See the vstiff(1) reference page for more help.

   Note: If your filter converts directly to *STIFF93aImage*, the PostScript options in *vstiff* are not applicable.

7. Finally, try printing that file to an Impressario printer. Enter:

```
lp -dprintername /usr/tmp/sample.blastfile
```

Watch the spooler log file for errors and the *printstatus* panel for printer messages until the file prints. If you were able to *vstiff* the file, then you should to be able to print it.

When you have completed this process, you should have the files shown in Table 11-1. These files should be installed with the permissions and the ownerships shown and at the locations shown. The actual file basename may change but the pathname should not. Conversion filters should be installed in */usr/lib/print*.

Table 11-1   New Filetype Pathnames

Mode	Owner	Group	Full Pathname
-r--r--r--	root	sys	/usr/lib/filetype/install/blastfiletype.ftr
-rwxr-xr-x	root	sys	/usr/lib/print/blast2stiff

Using an Alternate PostScript RIP

Using an alternate PostScript Raster Image Processor (RIP) is extremely easy in the Impressario open architecture. You can even switch RIPs at the last possible moment without losing any of the features of the Impressario file conversion pipeline.

The steps required to add an alternate RIP to your system are as follows:

1. Make the new RIP command line compatible with the Impressario RIP, psrip.
2. Write a dummy TYPE.
3. Test the alternative RIP.
4. Package the files.

You should be able to use this alternate RIP instead of, or in addition to, the standard Impressario interpreter.

Note: In the example below, the new RIP is called *blastrip* and is installed in the directory */usr/lib/print*.

Making the Command Line Compatible With psrip

Your new RIP must be command-line compatible with psrip. Look at the psrip(1) reference page for more information on what that means. psrip accepts ASCII and binary PostScript either on standard in or in a disk file, and generates a variety of sizes, colorspaces, and depths of STIFF bitmaps on standard out.

Being command-line compatible with psrip is probably the most tedious part of adding a new interpreter, but is well worth the effort when you consider that you gain all the advantages of Impressario:

- plug-and-play printer drivers
- automatic file type recognition
- automatic file type conversion

Writing a Dummy TYPE

The next step is to write a dummy TYPE and a simple CONVERT rule. First create a new, empty TYPE definition for your output raster format. For example:

```
TYPE BlastRasterBitmap
# dummy type for use in model file only
    MATCH false;

CONVERT ImpressarioPostScriptFile BlastRasterBitmap
# convert from formatted, filtered ImpressarioPostScriptFile
# to a RIPped bitmap
    COST   50
    FILTER /usr/lib/print/blastrip $IMPR_PSRIPOPTS
```

Install that file in */usr/lib/filetype/install/blastrip.ftr* and compile your FTR database by entering

su ; cd /usr/lib/filetype ; make

Now test the conversion and preview the bitmap by using *vstiff*:

fileconvert -d BlastRasterBitmap /etc/passwd | vstiff -stdin

If you run into problems viewing it, then save it to disk and use other tools to verify that you have a valid STIFF file. However, if *vstiff* cannot view it, then Impressario printer drivers will probably not be able to print it.

Testing the Alternate RIP

To use the alternative RIP in your printer model files instead of the default psrip, change all references to "ImpressarioRasterBitmap" to "BlastRasterBitmap." The lines below were taken from the *laserjetPJL_model* template model file and the word *ImpressarioRasterBitmap* was changed to *BlastRasterBitmap*:

```
# Use fileconvert to convert to the printer's native format.
# For raster printers, it is BlastRasterBitmap.
# For PostScript printers, it is ImpressarioPostScriptFile.
#
CMD='$fileconvert $fileconvertopts –d BlastRasterBitmap $file 2> /dev/null'
```

Packaging the Files

You are finished now, and need only package these optional RIP files for installation. The files you should be installing may have different names, but should be installed in the following directories, with the permissions and ownerships shown in Table 11-2.

Table 11-2   Alternative RIP Pathnames

Mode	Owner	Group	Full Pathname
-r--r--r--	root	sys	/usr/lib/filetype/install/blastrip.ftr
-rwxr-xr-x	root	sys	/usr/lib/print/blastrip

You should modify your printer's model file to use the new RIP whenever you want. You can, of course, still ask for *ImpressarioRasterBitmap*.

Note: Test carefully to make sure that your new RIP is, in fact, compatible with the existing psrip before shipping your product.

Stream TIFF Data Format

This appendix describes the Stream TIFF file format (STIFF), the primary interchange file format between printer filters and drivers. It also describes libstiff, a C-language application programming interface (API) used to read and write Stream TIFF files.

Stream TIFF Data Format

This appendix describes the Stream TIFF data format (STIFF), the primary interchange file format between printer filters and drivers, and *libstiff*, a C application program interface (API) used to read and write Stream TIFF files. Stream TIFF is also used by *gscan* to store images in TIFF files and to scan to the screen (in conjunction with *vstiff*).

The following major topics are discussed:

Library Description

libstiff provides a C application program interface (API) to the reading and writing of Stream TIFF files. Stream TIFF (STIFF) is a subset of the Tag Image File Format (TIFF) Revision 6.0 originally developed by Aldus Corporation. TIFF is an extremely flexible format, well suited for monochrome and color bitmap images. The primary difference between TIFF and STIFF is that while a TIFF file may require file seeking during reading or writing, a STIFF file does not. This means that a STIFF file can be read and written to both files and non-seekable streams such as pipes and sockets. STIFF can be used by application developers to write TIFF 6.0-compliant files, and a STIFF file can be read by any TIFF reader that conforms to the TIFF Revision 6.0 specifications. However, *libstiff* cannot be used to read many standard TIFF files since STIFF is a subset of TIFF.

The functions provided by *libstiff* greatly simplify the reading and writing of TIFF-compatible files. Using the STIFF API, TIFF 6.0-compatible STIFF files can be read and written without the need to understand the structure of a TIFF file and without the need to explicitly specify TIFF tags.

Library Access

There are two sets of *libstiff* functions. One set comprises the generic *libstiff* API. These functions are designated by an ST prefix and may be used to read and write generic STIFF files. To access these functions, an application must include the header file *stiff.h* located in the */usr/include* directory. The second set of *libstiff* functions is tailored to reading and writing STIFF files that are to be passed between printing filters and drivers. These printing-related functions are designated by the prefix PST and are accessed through the header file *printstiff.h*, also located in */usr/include*. If *printstiff.h* is used, the header *stiff.h* need not be specified. The generic and printing-specific functions may be freely intermixed within an application.

Programs that call *libstiff* functions must link with the *libstiff.a* library located in the directory */usr/lib*. An example command line is shown below:

```
cc -o myprog myprog.c -lstiff
```

Library Functions

libstiff provides the generic functions listed in Table A-1.

Table A-1   STIFF Generic Functions

Task	Function Name	Description
Stream Handling	STOpen()	Opens a STIFF stream on the specified file descriptor
	STClose()	Closes a STIFF stream opened by STOpen()
	STSkipTo()	Skips forward on a specified stream
Reading and Writing	STReadImageHeader()	Reads the STIFF image header from the specified stream
	STWriteImageHeader()	Writes the image header to the specified stream
	STRead()	Reads the specified amount of image data from the stream
	STWrite()	Writes the specified amount of image data to the stream
TIFF Tag Support	STAddTag()	Adds the specified tag to the TIFF tag list
	STRemoveTag()	Removes the specified tag from the output TIFF tag list
	STGetTag()	Searches the tag list for the specified TIFF tag
	STPrintTags()	Prints the current TIFF tag list
Execution Error Handling	STPerror()	Prints error messages to standard error
	STErrorString()	Gets the error string for the specified error code

STIFF also provides the printing-specific functions listed in Table A-2.

Table A-2   STIFF Printing-Specific Functions

Function Name	Description
PSTReadImageHeader()	Reads the printing-specific STIFF image header from the stream
PSTWriteImageHeader()	Writes the printing-specific STIFF image header to the stream

Example Usage

The sequence of operations for writing a STIFF stream is as follows:

1. Obtain a writable file descriptor. Note that this descriptor can be associated with a non-seekable stream.

2. Call **STOpen()** with the writable file descriptor and the flag ST_WRITE.

3. Fill in the STIFF image header (**STImageHeader()** or **PSTImageHeader()**).

4. Optionally, add any application-specific TIFF tags to the file using **STAddTag()**.

5. Call the image header write function **STWriteImageHeader()** or **PSTWriteImageHeader()**.

6. Write the image data using **STWrite()**.

7. Repeat steps 3 through 6 for each page of image data.

8. Close the STIFF file using **STClose()**.

9. Close the file descriptor.

The sequence of operations for reading a STIFF stream is as follows:

1. Obtain a readable file descriptor. Note that this descriptor can be associated with a non-seekable stream.

2. Call **STOpen()** with the readable file descriptor and the flag ST_READ.

3. Call the image header read function **STReadImageHeader()** or **PSTReadImageHeader()**.

4. Access the fields of the image header structure to determine the amount of image data to be read for this page.

5. Optionally, retrieve any application-specific TIFF tags from the file using **STGetTag()**.

6. Read the image data using **STRead()**.

7. Repeat steps 3 through 6 for each page of image data. The last page of image data is an empty page (that is, the amount of data equals zero). An STEEOF error occurs if an attempt is made to read past the end of the STIFF.

8. Close the STIFF stream using **STClose()**.

9. Close the file descriptor.

If an error condition is returned by a *libstiff* function, **STPerror()** can be used to print a diagnostic error message to the standard error, and **STErrorString()** can be used to obtain an appropriate error message for display to the user by other means.

Printing-Specific STIFF

The printing-specific functions (PST) provided by *libstiff* read and write STIFF files as described above. The printing-specific aspect of these functions is found in the image header structure. **PSTImageHeader()**, the printing-specific image header, contains all fields of the **STImageHeader()** plus printing-specific information fields such as image resolution and halftoning method.

If a STIFF file is written using the generic functions and is read using the printing-specific functions, default values are used for the **PSTImageHeader()** fields not found in the STIFF. Similarly, a STIFF file written using the printing-specific functions can be read by the generic functions. In this case, the additional information in the stream can be ignored or obtained using the **STGetTag()** function.

Refer to the header file */usr/include/printstiff.h* for additional information regarding printing-specific STIFF.

Printer driver developers should pay special attention to the command-line options string that is part of a PST image header. See the LaserJet example driver for an example of how to combine parsing the command-line and image-header driver options.

Generic STIFF File Structure

While it is not necessary to understand the STIFF file structure to use *libstiff*, this explanation is provided for those who need to know. A Stream TIFF file is first and foremost a valid TIFF file. STIFF is derived from the TIFF 6.0 specifications available from Aldus Corporation (see below). The terms used below to describe a STIFF file (for example, IFD) are explained in the TIFF specifications and are not described here.

The primary restriction STIFF places on the TIFF structure is that all data must be read from or written to the file without the need to seek within the file. Specifically, within a STIFF file, these things must be true:

1. The bitmap image data must be in page-number order.

2. Data that does not fit in the value section of a tag must be located immediately after the IFD and must occur in the same order in which the tags are encountered. The exception to this is the image data itself, which must come last for each page.

3. Image data must immediately follow the IFD and any associated offset values.

4. A terminating, empty IFD is always added to the end of the STIFF file. This IFD guarantees that an IFD with 0000 in its "next IFD" field appears in the IFD chain. Note that this empty IFD is not encountered when following IFD pointers if the last "real" IFD is written with the *last* parameter set to 1. While the TIFF specification states that IFDs should not be empty, relaxing this restriction appears to have no impact on TIFF compatibility.

A generic STIFF file can be represented as shown in Figure A-1.

TIFF header (8 bytes)
Optional intervening space
IFD for page 1
Data for long values in IFD 1
Image data for page 1
Optional intervening space
IFD for page 2
Data for long values in IFD 2
Image data for page 2
• • •
Terminating empty IFD
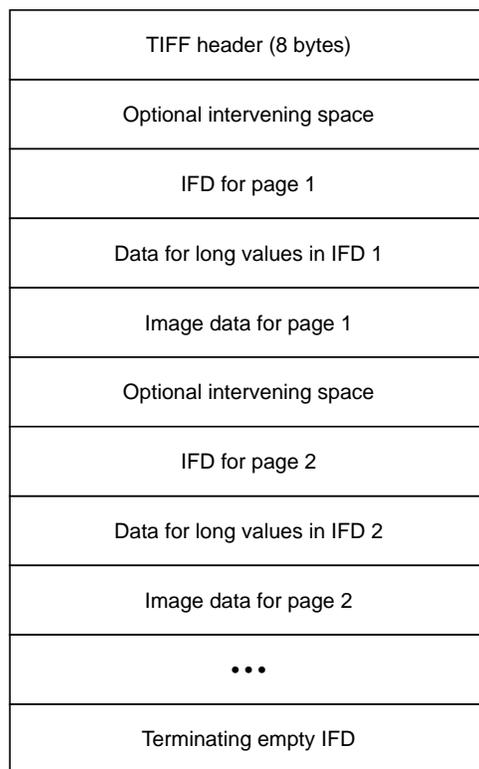
Figure A-1   Generic STIFF File Structure

The *libstiff* functions support only the Motorola® (MM) byte ordering. In addition to supporting TIFF class RGB data, *libstiff* supports the CMYK color image data type (*PhotoMetricInterpretation* = 5 and *InkSet* = 1) and four additional color image separation types: CMY, YMC, YMCK, and KCMY.

For these additional types, *PhotoMetricInterpretation* = 5, *InkSet* = 2, *NumberOfInks* = 3 or 4, and the *InkNames* tag is used to indicate the inks contained in each channel.

When reading an image header, *libstiff* parses the ink names for these additional types and sets the type field of the **STImageHeader** structure to the appropriate value defined in *stiff.h.* When writing an image header, *libstiff* writes the appropriate *PhotoMetricInterpretation*, *InkSet*, *NumberOfInks*, and *InkNames* tags based on the value of the type field of the **STImageHeader** structure.

The CMYK data format is a TIFF data format extension, as shown in Table A-3.

Table A-3   CMYK Data Format

Tag	Possible Values
BitsPerSample	(1,1,1,1) (4,4,4,4) (8,8,8,8)
PhotoMetricInterpretation	5
SamplesPerPixel	4
PlanarConfiguration	1, 2
NumberOfInks	4

The CMY data class is a subset of the CMYK class and differs from the CMYK class in a TIFF-compliant manner, as shown in Table A-4.

Table A-4   CMY Data Format

Tag	Possible Values
BitsPerSample	(1,1,1,1) (1,1,1) (4,4,4) (8,8,8)
PhotoMetricInterpretation	5
SamplesPerPixel	3 (4 or 8 bits, 1-bit planar) or 4 (1-bit chunky)
PlanarConfiguration	1, 2
NumberOfInks	3

The YMC data class is similar to the CMY class except that data is organized as YMC instead of CMY (see Table A-5). When using *libstiff*, YMC data corresponds to the data type **ST_TYPE_YMC**.

Table A-5   YMC Data Format

Tag	Possible Values
BitsPerSample	(1,1,1,1) (1,1,1) (4,4,4) (8,8,8)
PhotoMetricInterpretation	5
SamplesPerPixel	3 (4 or 8 bits, 1-bit planar) or 4 (1-bit chunky)
PlanarConfiguration	1, 2
NumberOfInks	3
InkSet	2
InkNames	yellow, magenta, cyan

The YMCK class is similar to the CMYK class except that data is organized as YMCK instead of CMYK (see Table A-6). When using *libstiff*, YMCK data corresponds to the data type **ST_TYPE_YMCK**.

Table A-6   YMCK Data Format

Tag	Possible Values
BitsPerSample	(1,1,1,1) (4,4,4,4) (8,8,8,8)
PhotoMetricInterpretation	5
SamplesPerPixel	4
PlanarConfiguration	1, 2
NumberOfInks	4
InkSet	2
InkNames	yellow, magenta, cyan, black

The KCMY class is similar to the CMYK class except that the data is organized as KCMY instead of CMYK (see Table A-7). When using *libstiff*, KCMY data corresponds to the data type **ST_TYPE_KCMY**.

Table A-7   KCMY Data Format

Tag	Possible Values
BitsPerSample	(1,1,1,1) (4,4,4,4) (8,8,8,8)
PhotoMetricInterpretation	5
SamplesPerPixel	4
PlanarConfiguration	1, 2
NumberOfInks	4
InkSet	2
InkNames	black, cyan, magenta, yellow

Note that for the RGB, CMY, and YMC classes with *BitsPerSample* values of (1,1,1) and a *PlanarConfiguration* value of 1, pixels are stored two to a byte, with the bits ordered from most-significant to least-significant.

For example, CMY data is stored as

CMY0CMY0

Rows are padded to contain an integral number of bytes.

Refer to the header file *stiff.h* for additional information regarding the STIFF file structure.

Appendix B

Silicon Graphics Image File Format API

This chapter describes libimp, the C-language application programming
interface (API) for reading and writing Silicon Graphics Image Format files.

Silicon Graphics Image File Format API

This appendix describes *libimp*, the C application program interface (API) for reading and writing Silicon Graphics image format files.

The following major topics are discussed:

- "Library Description" on page 167
- "Library Access" on page 167
- "Library Functions" on page 168
- "IMPImage Structure" on page 171

Library Description

libimp provides a C application program interface (API) for reading and writing Silicon Graphics image format files and for performing a number of format-independent image processing operations. These operations include color space conversion and filtered image zooming.

libimp provides all functionality of the *libimage* library. (See the rgb(4) reference page for a description of *libimage*.) In addition, *libimp* provides function prototypes, a documented interface, reliable error reporting, and a number of other enhancements.

Library Access

A program that calls *libimp* functions must include the header file *imp.h* located in the directory */usr/include*. In addition, the program must link with the *libimp.a* library located in */usr/lib*. The link line would look like this:

```
... -limp ...
```

Library Functions

The *libimp* library, which is based heavily on the *libimage* and *libgutil* libraries, consists of two main sets of functions. The functions shown in Table B-1 perform operations on Silicon Graphics image format files.

Table B-1   Silicon Graphics Image Format File Functions

Task	Function	Description
Image Access	impOpen()	Opens a Silicon Graphics image format file for reading or writing
	impOpenFd()	Opens a Silicon Graphics image format file for reading or writing
	impClose()	Closes a Silicon Graphics image format file
	impCloseFd()	Closes a Silicon Graphics image format file
Image I/O	impReadRow()	Reads image row
	impReadRowB()	Reads byte image row
	impWriteRow()	Writes image row
	impWriteRowB()	Writes byte image row

The functions shown in Table B-2 perform operations on image data in a format-independent manner.

Table B-2   Format-Independent File Functions

Task	Function	Description
Zooming	impCreateZoom()	Creates zoom operator
	impDestroyZoom()	Destroys zoom operator
	impResetZoom()	Resets zoom row cache
	impZoomRow()	Zooms an image row
Data Packing	impPackRow()	Packs two-byte data into one byte
	impUnpackRow()	Unpacks one-byte data into two bytes
Math Operations	impZeroRow()	Sets row to zero
	impInitRow()	Initializes a row to a value
	impCopyRow()	Copies a row
	impSAddRow()	Adds a value to a row
	impVAddRow()	Adds two rows
	impSSubRow()	Subtracts a value from a row
	impVSubRow()	Subtracts two rows
	impSMulRow()	Multiplies a row by a value
	impSDivRow()	Divides a row by a value
	impClampRow()	Clamps row values
Color Space Conversion	impRGBtoW()	Converts an array from RGB to W format
	impWtoRGB()	Converts an array from W to RGB format
	impRGBtoK()	Converts an array from RGB to K format
	impKtoRGB()	Converts an array from K to RGB format
	impRGBtoCMY()	Converts an array from RGB to CMY format
	impCMYtoRGB()	Converts an array from CMY to RGB format

Table B-2 (continued)   Format-Independent File Functions

Task	Function	Description
	impRGBtoYIQ()	Converts an array from RGB to YIQ format
	impYIQtoRGB()	Converts an array fromYIQ to RGB format
	impRGBtoYUV()	Converts an array from RGB to YUV format
	impYUVtoRGB()	Converts an array from YUV to RGB format
	impRGBtoYCbCr()	Converts array from RGB to YCbCr format
	impYCbCrtoRGB()	Converts array from YCbCr to RGB format
	impRGBtoCMYK()	Converts array from RGB to CMYK format
	impCMYKtoRGB()	Converts array from CMYK to RGB format
	impRGBtoDevCMYK()	Converts array from RGB to device CMYK format
	impRGBtoHSV()	Converts an array from RGB to HSV format
	impHSVtoRGB()	Converts an array from HSV to RGB format
	impRGBtoHLS()	Converts an array from RGB to HLS format
	impHLStoRGB()	Converts an array from HLS to RGB format
Error Handling	impPerror()	Prints *libimp* execution error messages to standard error
	impErrorString()	Obtains *libimp* execution error messages

IMPImage Structure

The **IMPImage** structure contains public and private information about a Silicon Graphics image file. This structure is identical both in size and field naming to the IMAGE structure defined in the header file *image.h*, included by applications that use the *libimage* library. While it has been common practice to directly modify the public fields of the image structure, this is not recommended. Macros are defined in *imp.h* for manipulating the structure fields. It is strongly recommended that these macros be used to set and get values from the image structure. The **IMPImage** structure is defined as follows:

```
typedef struct _impImage {
    /******* Public image header information (archived) */
    ushort_t imagic;  /* Silicon Graphics image file magic number */
    ushort_t type;     /* Raster type (e.g. verbatim, rle) */
    ushort_t dim;      /* Image dimension */
    ushort_t xsize;    /* X size (pixels) */
    ushort_t ysize;    /* Y size (pixels) */
    ushort_t zsize;    /* Number of channels (e.g. rgb = 3) */
    long     min;      /* Minimum intensity in image */
    long     max;      /* Maximum intensity in image */
    ulong_t  wastebytes;          /* Padding */
    char     name[IMP_NAME_MAX+1];  /* Image name */
    ulong_t  colormap;            /* Image type (e.g. colormap, normal) */

    /******* Private image header information (core use only) */
    long     file;
    ushort_t flags;
    short    dorev;
    short    x;
    short    y;
    short    z;
    short    cnt;
    short    *ptr;
    short    *base;
    short    *tmpbuf;
    ulong_t  offset;
    ulong_t  rleend;
    ulong_t  *rowstart;
    long     *rowsize;
} IMPImage;
```

Note: *ushort_t* and *ulong_t* are unsigned short and unsigned long, respectively.

171

Fields:

magic	Magic number identifying file as a Silicon Graphics image format file.
type	Bitwise-OR combined code indicating the raster encoding method and the number of bytes per pixel per channel. Currently, Silicon Graphics image files support either a verbatim, uncompressed raster encoding or a run-length, compressed encoding. Both of these encodings are available at one or two bytes per pixel per channel. The header file *imp.h* defines codes for all supported combinations of encoding methods and pixel widths.
dim	Number of dimensions to the image. A colormap file has dimension one (length), a black and white image has dimension two (height and width), and an RGB image has dimension three (height, width, and depth).
xsize, *ysize*	Image size in pixels.
zsize	Number of color channels or depth. A black and white image has one channel and an RGB image has three channels.
min, *max*	The minimum and maximum intensity values in the image. These values are the minimum and maximum for all channels combined.
name	A descriptive name string for the image.
colormap	The image type. Refer to *imp.h* for the supported image type codes. The field is named colormap for compatibility with the **IMAGE** structure used by the *libimage* library.

Image Access Functions

impOpen() Function

This function opens the image file specified by *fname*. If *mode* is *r*, the file is opened for reading. If *mode* is *w*, the file is opened for writing and created if it does not exist, or truncated to zero length if it does exist.

impOpenFd() opens the image file pointed to by the file descriptor *fd*. The descriptor's permissions must permit the operations specified by *mode*. That is, if *mode* is *w*, the descriptor must have write permission. In addition, it must be possible to seek on the specified descriptor. At this time, read/write mode is not supported for Silicon Graphics image files. Upon successful execution, both functions return a pointer to a Silicon Graphics image file structure.

Synopsis:

```
#include <imp.h>

IMPImage* impOpen(const char *fname, const char *mode, ...);
IMPImage* impOpenFd(int fd, const char *mode, ...);
```

In write mode, **impOpen()** and **impOpenFd()** require that these additional parameters be specified:

```
uint_t rasterType,dimension, xSize, ySize, numChannels, imageType;
char *name;
```

Note: *uint_t* stands for unsigned int.

Arguments:

rasterType	Specifies the raster encoding method and the number of bytes per pixel per channel. Silicon Graphics image format files can be written either uncompressed or with run-length encoding compression, and with one or two bytes per pixel per channel. Refer to *imp.h* for the supported raster types.
dimension	Specifies the number of dimensions in the image. A colormap file has dimension one, a black and white image has dimension two, and an RGB image has dimension three.
xSize, *ySize*	Specifies the image size in pixels.

numChannels	Specifies the number of image color channels. A black and white image has one channel and an RGB image has three channels.
imageType	Specifies how the image data is to be interpreted. Image data is either actual color values (normal), screen colormap indices (screen), or a colormap (colormap). Refer to *imp.h* for the supported image types.
name	Specifies a descriptive string for the image. Strings longer than IMP_NAME_MAX characters are truncated. Refer to *imp.h* for the value of IMP_NAME_MAX. If this parameter is specified as NULL, the string "no name" is written into the file. Use the empty string "" to write an empty name string into the image.

impOpen(), impOpenFd(), impClose(), and impCloseFd() Functions

impClose() closes a Silicon Graphics image format file previously opened by **impOpen()** or **impOpenFd()**. Among other tasks, **impClose()** closes the file descriptor associated with an image file. If the image was opened using **impOpenFd()**, the file descriptor specified in that function call is closed by **impClose()**.

impCloseFd() performs the same function as **impClose()**, but it leaves open the file descriptor associated with the image and returns it in the parameter *fdp*. It then becomes the responsibility of the caller to close the file descriptor when it is no longer needed. It is essential that either **impClose()** or **impCloseFd()** be called at the completion of writing a Silicon Graphics image file so that all buffered data can be written and the image header can be updated.

Synopsis:

```
#include <imp.h>

int impClose(IMPImage *image);
int impCloseFd(IMPImage *image, int *fdp);
```

Return Value:

impOpen() and **impOpenFd()** return a pointer to an image structure if execution was successful. NULL is returned and *IMPerrno* is set if an execution error has occurred.

impClose() and **impCloseFd()** return 0 if execution was successful; a -1 is returned and *IMPerrno* is set if an execution error has occurred.

Execution Error Codes:

impOpen() and **impOpenFd()** fail with the following errors:

- IMP_ERR_READWRITE
- IMP_ERR_MEMALLOC
- IMP_ERR_BADMAGIC
- IMP_ERR_BADRASTER
- IMP_ERR_BADIMAGE

In addition, **impOpenFd()** fails with the following errors:

- IMP_ERR_BADFD
- IMP_ERR_SEEK

impClose() and **impCloseFd()** fail with the following errors:

- IMP_ERR_WRITEFLAG
- IMP_ERR_BADBPP
- IMP_ERR_BADIMAGE

Note: The storage for the **IMPImage** structure is allocated by the image open function. This storage is deallocated by the **impClose()** and **impCloseFd()** functions. The caller should not explicitly reallocate or deallocate any storage related to the image structure.

See also:

libimp(3), impReadRow(3), impReadRowB(3)

Data Packing Functions

impPackRow() and impUnpackRow() Functions

Synopsis:

```
#include <imp.h>

void impPackRow(uchar_t *dptr, short *sptr, int n);
void impUnpackRow(short *dptr, uchar_t *sptr, int n);
```

impPackRow() converts the array of short integers pointed to by *sptr* into the array of unsigned **char** values pointed to by *dptr*. Source data that is too large to fit in a character is truncated. For example, the source value 0x0B56 is converted into 0x56 in the destination array. **impUnpackRow()** converts the array of unsigned **char** values pointed to by *sptr* into the array of short integers pointed to by *dptr*. For example, the source value 0x56 is converted into 0x0056 in the destination array. The parameter *n* specifies the number of elements in the source and destination arrays.

Note: The allocation of storage for the source and destination arrays is the responsibility of the caller.

See also:

libimp(3)

Error Handling Functions

impPerror() and impErrorString() Functions

Synopsis:

```
#include <imp.h>

void impPerror(const char *str);
char* impErrorString(int errCode);
extern int IMPerrno;
```

impPerror() prints error messages to standard error in a format similar to the standard C library function perror(3C). If an error occurs during a *libimp* function call, the global error variable *IMPerrno* is set with an error code. The error code is either a system error code (*errno*) or a *libimp*-specific code. The symbolic names for the *libimp* error codes are defined in *imp.h*. The value of *IMPerrno* is used by **impPerror()** as an index to a table of error messages.

impPerror() prints user-supplied string *str* followed by a colon (:), a space, and the error message corresponding to the current value of *IMPerrno*.

If the string *str* is the NULL string (""), no colon or space is printed, only the error message. The error message is either a system error message or a *libimp*-specific message. To be of most use, a call to *impPerror* should be made immediately following the *libimp* function call where an error has been detected. **impErrorString()** is similar to the strerror(3C) function and returns the error message corresponding to the error code specified by *errCode*.

If *errCode* is less than IMP_ERR_BASE, the message returned is a system error message generated by *strerror*. If *errCode* is one of the error codes specified in *imp.h*, the returned string is a *libimp*-specific error message.

See also:

libimp(3), perror(3C), strerror(3C)

Image I/O Functions

impReadRow(), impReadRowB(), impWriteRow(), and impWriteRowB() Functions

Synopsis:

```
#include <imp.h>

int impReadRow(IMPImage *image, short *buffer,
   ushort_t row, ushort_t channel);
int impReadRowB(IMPImage *image, uchar_t *buffer,
   ushort_t row, ushort_t channel);

int impWriteRow(IMPImage *image, short *buffer,
   ushort_t row, ushort_t channel);
int impWriteRowB(IMPImage *image, uchar_t *buffer,
   ushort_t row, ushort_t channel);
```

impReadRow() and **impReadRowB()** each read a row of image data from the specified channel of a Silicon Graphics image format file.

impReadRow() stores the row data in an array of short integers and can read image data that is one or two bytes per pixel per channel in width. **impReadRowB()** stores the data in a character array and can handle only image data that is one byte per pixel per channel wide.

If **impReadRowB()** is called to read image data that is two bytes per pixel per channel, an error condition is reported. **impWriteRow()** and **impWriteRowB()** each write a row of image data to the specified channel of a Silicon Graphics image file. **impWriteRow()** writes one or two bytes per pixel per channel image row data. **impWriteRowB()** writes only one byte per pixel data. It is an error to use **impWriteRowB()** to write to images that expect two bytes per pixel per channel data.

Note: The functions make use of the following macros:

impXSize   Returns the number of pixels in an image file in the X direction.

impYSize   Returns the number of pixels in an image file in the Y direction.

impNumChannels

Returns the number of color channels in an image. It returns 3 if the image is RGB, 4 if it is CMYK, and 1 if it is monochrome

The functions take the following parameters:

image   Pointer to an **IMPImage** structure returned by a call to **impOpen()** or **impOpenFd()**.

buffer   Caller-allocated buffer containing the data to write to or to be filled with the data read from the image. The amount of storage allocated for the buffer should be **impXSize(**image**)** x **sizeof(**short**)** if **impReadRow()** or **impWriteRow()** is used, and **impXSize(**image**)** if **impReadRowB()** or **impWriteRowB()** is used.

row   The image row to read. Rows are numbered from 0 through **impYSize(**image**)** minus 1.

channel   The image channel to read. Channels are numbered from 0 through **impNumChannels(**image**)** minus 1.

Return Value:

If execution was successful, all functions return the number of pixels (not bytes) read or written. If an execution error occurred, -1 is returned and *IMPerrno* is set.

Execution Error Codes:

impWriteRow() and **impWriteRowB()** fail with the following errors:

- IMP_ERR_WRITEFLAG
- IMP_ERR_BADBPP
- IMP_ERR_BADIMAGE
- IMP_ERR_SHORTWRITE

179

impReadRow() and **impReadRowB()** fail with the following errors:

- IMP_ERR_READFLAG
- IMP_ERR_BADBPP
- IMP_ERR_BADIMAGE
- IMP_ERR_SHORTREAD

Note: It is the caller's responsibility to allocate enough buffer storage for image row data.

See also:

libimp(3), impOpen(3)

Color Space Conversion Functions

These functions perform color space conversion between a given color space and RGB. The actual transformations performed are described below. Certain functions specify the parameter *unity*. *unity* should be set to the value of maximum possible intensity for the arrays specified. For example, if 8-bit data is being converted, unity would be specified as 255. If the data makes use of the full 16 bits available in each array element, *unity* would be specified as 65535. Note that the parameter *n* specifies the number of elements in the arrays and not the number of bytes.

impRGBtoW(), impWtoRGB() Functions

Figure B-1 shows the equation for W conversions.

$$W = \begin{vmatrix} 0.299 & 0.587 & 0.114 \end{vmatrix} * \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} W \\ W \\ W \end{vmatrix}$$

Figure B-1   W Conversions

Synopsis:

```
#include <imp.h>

void impRGBtoW(short *rbuf, short *gbuf, short *bbuf,
   short *wbuf, int n);

void impWtoRGB(short *wbuf, short *rbuf, short *gbuf,
    short *bbuf, int n);
```

impRGBtoK(), impKtoRGB() Functions

Figure B-2 shows the equation for K conversions.

$$K = \begin{vmatrix} 0.299 & 0.587 & 0.114 \end{vmatrix} * \begin{vmatrix} 1.0 - R \\ 1.0 - G \\ 1.0 - B \end{vmatrix}$$

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} unity - K \\ unity - K \\ unity - K \end{vmatrix}$$

Figure B-2   K Conversions

Synopsis:

```
#include <imp.h>

void impRGBtoK(short *rbuf, short *gbuf, short *bbuf,
   short *kbuf, short unity, int n);

void impKtoRGB(short *kbuf, short *rbuf, short *gbuf,
    short *bbuf, short unity, int n);
```

impRGBtoCMY(), impCMYtoRGB() Functions

Figure B-3 shows the equation for CMY conversions.

$$\begin{vmatrix} C \\ M \\ Y \end{vmatrix} = \begin{vmatrix} 1.0 - R \\ 1.0 - G \\ 1.0 - B \end{vmatrix}$$

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1.0 - C \\ 1.0 - M \\ 1.0 - Y \end{vmatrix}$$

Figure B-3   CMY Conversions

Synopsis:

```
#include <imp.h>

void impRGBtoCMY(short *rbuf, short *gbuf, short *bbuf,
    short *cbuf, short *mbuf, short *ybuf,
    short unity, int n);

void impCMYtoRGB(short *cbuf, short *mbuf, short *ybuf,
    short *rbuf, short *gbuf, short *bbuf,
    short unity, int n);
```

impRGBtoYIQ(), impYIQtoRGB() Functions

Figure B-4 shows the equation for YIQ conversions.

$$\begin{vmatrix} Y \\ I \\ Q \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.212 & -0.523 & 0.311 \end{vmatrix} * \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1.000 & 0.955 & 0.622 \\ 1.000 & -0.271 & -0.648 \\ 1.000 & -1.107 & 1.702 \end{vmatrix} * \begin{vmatrix} Y \\ I \\ Q \end{vmatrix}$$

Figure B-4   YIQ Conversions

Synopsis:

```
#include <imp.h>

void impRGBtoYIQ(short *rbuf, short *gbuf, short *bbuf,
   short *ybuf, short *ibuf, short *qbuf,
   int n);

void impYIQtoRGB(short *ybuf, short *ibuf, short *qbuf,
   short *rbuf, short *gbuf, short *bbuf,
   int n);
```

impRGBtoYUV(), impYUVtoRGB() Functions

Figure B-5 shows the equation for YUV conversions.

$$
\begin{vmatrix} Y \\ U \\ V \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{vmatrix} * \begin{vmatrix} R \\ G \\ B \end{vmatrix}
$$

$$
\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1.000 & 0.000 & 1.140 \\ 1.000 & -0.395 & -0.581 \\ 1.000 & 2.032 & 0.000 \end{vmatrix} * \begin{vmatrix} Y \\ U \\ V \end{vmatrix}
$$

Figure B-5   YUV Conversions

Synopsis:

```
#include <imp.h>

void impRGBtoYUV(short *rbuf, short *gbuf, short *bbuf,
   short *ybuf, short *ubuf, short *vbuf,
   int n);

void impYUVtoRGB(short *ybuf, short *ubuf, short *vbuf,
   short *rbuf, short *gbuf, short *bbuf,
   int n);
```

impRGBtoYCbCr(), impYCbCrtoRGB() Functions

Figure B-6 shows the equation for YCbCr conversions.

$$\begin{vmatrix} Y \\ Cb \\ Cr \end{vmatrix} = \begin{vmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{vmatrix} * \begin{vmatrix} R \\ G \\ B \end{vmatrix}$$

$$\begin{vmatrix} R \\ G \\ B \end{vmatrix} = \begin{vmatrix} 1.000 & -0.001 & 1.402 \\ 1.000 & -0.344 & -0.714 \\ 1.000 & 1.772 & 0.001 \end{vmatrix} * \begin{vmatrix} Y \\ Cb \\ Cr \end{vmatrix}$$

Figure B-6   YCbCr Conversions

Synopsis:

```
#include <imp.h>

void impRGBtoYCbCr(short *rbuf, short *gbuf, short *bbuf,
    short *ybuf, short *cbbuf, short *crbuf, int n);

void impYCbCrtoRGB(short *ybuf, short *cbbuf, short *crbuf,
    short *rbuf, short *gbuf, short *bbuf, int n);
```

impRGBtoCMYK(), impRGBtoDevCMYK(), impCMYKtoRGB() Functions

Figure B-7 shows the equation for CMYK conversions.

Ci	=	1.0 - R	Ci	=	C + K
Mi	=	1.0 - G	Mi	=	M + K
Yi	=	1.0 - B	Yi	=	Y + K
K	=	Min(Ci, Mi, Yi)	R	=	1.0 - Min(1.0, Ci)
C	=	Ci - K	G	=	1.0 - Min(1.0, Mi)
M	=	Mi - K	B	=	1.0 - Min(1.0, Yi)
Y	=	Yi - K			

Ci	=	1.0 - R
Mi	=	1.0 - G
Yi	=	1.0 - B
Ki	=	Min(Ci, Mi, Yi)
Kucr	=	UCR(Ki)
Device M	=	Min(1.0, Max(0.0, Ci - Kucr))
Device C	=	Min(1.0, Max(0.0, Mi - Kucr))
Device Y	=	Min(1.0, Max(0.0, Yi - Kucr))
Device K	=	BG(Ki)

Figure B-7   CMYK Conversions

Synopsis:

```
#include <imp.h>

void impRGBtoCMYK(short *rbuf, short *gbuf, short *bbuf,
   short *cbuf, short *mbuf, short *ybug,
   short *kbuf, short unity, int n);

void impRGBtoDevCMYK(short *rbuf, short *gbuf, short *bbuf,
   short *cbuf, short *mbuf, short *ybug,
   short *kbuf, IMPUCRFunc ucr, IMPBGFunc bg,
   short unity, int n);

short (*IMPBGFunc)(short k);
short (*IMPUCRFunc)(short k);

void impCMYKtoRGB(short *cbuf, short *mbuf, short *ybuf,
   short *kbuf, short *rbuf, short *gbug,
   short *bbuf, short unity, int n);
```

impRGBtoHSV(), impHSVtoRGB() Functions

Synopsis:

```
#include <imp.h>

void impRGBtoHSV(short *rbuf, short *gbuf, short *bbuf,
    float *hbuf, float *sbuf, float *vbuf, int n);

void impHSVtoRGB(float *hbuf, float *sbuf, float *vbuf,
    short *rbuf, short *gbuf, short *bbuf, int n);
```

impRGBtoHLS(), impHLStoRGB() Functions

For HSV conversions, refer to *Computer Graphics, Principals and Practice*, Foley and Van Dam, 2nd Edition, pages 590-592. For HLS conversions, refer to pages 592-595.

Synopsis:

```
#include <imp.h>

void impRGBtoHLS(short *rbuf, short *gbuf, short *bbuf,
    float *hbuf, float *lbuf, float *sbuf,
    short unity, int n);

void impHLStoRGB(float *hbuf, float *lbuf, float *sbuf,
    short *rbuf, short *gbuf, short *bbuf,
    short unity, int n);
```

Note: It is the caller's responsibility to allocate all buffer storage.

See Also:

libimp(3)

Math Operation Functions

impZeroRow(), impInitRow(), impCopyRow(), impSAddRow(), impVAddRow(), impSSubRow(), impVSubRow(), impSMulRow(), impSDivRow(), impClampRow() Functions

Synopsis:

```
#include <imp.h>

void impZeroRow(short *dptr, int n);

void impInitRow(short *dptr, int val, int n);

void impCopyRow(short *dptr, short *sptr, int n);

void impSAddRow(short *dptr, short *sptr, int val, int n);

void impVAddRow(short *dptr, short *sptr1, short *sptr2, int n);

void impSSubRow(short *dptr, short *sptr, int val, int n);

void impVSubRow(short *dptr, short *sptr1, short *sptr2, int n);

void impSMulRow(short *dptr, short *sptr, int val, int n);

void impSDivRow(short *dptr, short *sptr, int val, int n);

void impClampRow(short *dptr, short *sptr, int lov, int hiv, int n);
```

In the following descriptions, the parameter *n* specifies the number of elements in an array and not the number of bytes in the array. In addition, functions that take a source array pointer and a destination array pointer can specify the same array as both a source and destination.

impZeroRow   Initializes to zero the array pointed to by *dptr*.

impInitRow   Initializes the array *dptr* to the value *val*.

impCopyRow   Copies the array *sptr* to the array *dptr*.

impSAddRow   Adds the value *val* to each element of the array *sptr* and stores the result in the array *dptr*.

impVAddRow   Adds the corresponding elements of *sptr1* and *sptr2* and stores the result in the array *dptr*.

impSSubRow   Subtracts the value *val* from each element of the array *sptr* and stores the result in the array *dptr*.

impVSubRow   Subtracts the corresponding elements of *sptr2* from those of *sptr1* and stores the result in the array *dptr*.

impSMulRow   Multiplies each element of the array *sptr* by *val* and stores the result in *dptr*.

impSDivRow   Divides each element of the array *sptr* by *val* and stores the result in *dptr*.

impClampRow   Clamps the values of the array *sptr* between the values *lov* and *hiv* inclusive. The result is stored in the array *dptr*.

Note: It is the caller's responsibility to allocate all buffer storage. Also, because the arrays referenced by these functions are short integer arrays, the caller should be aware of overflow/wraparound conditions.

See also:

libimp(3)

Zooming Functions

impCreateZoom(), impDestroyZoom(), impResetZoom(), impZoomRow() Functions

Synopsis:

```
#include <imp.h>

IMPZoom* impCreateZoom(ushort_t srcXSize, ushort_t srcYSize,
   ushort_t dstXSize, ushort_t dstYSize,
   IMPReadRowFunc readRowFunc,
   int numChannels,
   IMPFilterType filterType,
   float blurFactor);

void impDestroyZoom(IMPZoom *zoom);

void impResetZoom(IMPZoom *zoom);

int impZoomRow(IMPZoom *zoom, short *buffer,
   ushort_t row, void *clientData);
```

The *libimp* library provides an API for performing image resizing or zooming. Images can be zoomed up or down using any of a number of resampling methods. The resampling methods divide into two categories. The first resampling category is non-filtered zooming (also known as replicative zoom, decimation). The second resampling category is filtered zooming where a filter of a given shape is applied to the data. The image zooming is performed on a row-by-row basis using a one-pass, two-dimensional convolution.

To zoom one or more rows of an image, first create a zoom operator by calling **impCreateZoom()**. One of the parameters to **impCreateZoom()** is a pointer to a function that is called during the zoom to read rows of the source image. To obtain zoomed rows, call **impZoomRow()**. When all desired zoomed rows have been obtained, call **impDestroyZoom()** to deallocate storage held by the zoom operator. When filtered zooming is performed, a number of contiguous rows of image data are cached. Often all rows of a given image channel are zoomed, followed by all rows of the next channel. Since rows are cached, the cache should be flushed when switching between image channels. The **impResetZoom()** function performs this row cache flushing operation.

The **impCreateZoom()** function has the following parameters:

srcXSize, srcYSize
Width and height of the source image in pixels.

dstXSize, dstYSize
Width and height of the destination (that is, zoomed) image in pixels.

readRowFunc   Pointer to a function that is called to read a row from the source image. The prototype for this function is

```
int (*IMPReadRowFunc)(short *buffer,
   ushort_t row,
   void *clientData);
```

The function should read the image row indicated by *row* and place the data in *buffer*. The storage for *buffer* is allocated and deallocated by the *zoom* operator and should not be manipulated by the read row function. *clientData* is a pointer to caller-specific information. The caller may specify a pointer to client data when calling the **impZoomRow()** function. That pointer is passed to the read row function.

At the caller's discretion, this pointer may be set to NULL. A common use of the client data is to pass a pointer to a structure containing the image structure pointer and the channel number. The read row function must return -1 if it encounters an error while obtaining the row data and must return a value of 0 or greater if the function succeeds.

numChannels   Specifies the number of channels of image data that are packed on a single row. For example, if each row contains data for only a single channel, then *numChannels* should be specified as one. However, if each row contains RGB data packed together as RGBRGBRGB..., *numChannels* should be specified as three.

filterType   Specifies the type of filter to be used for resampling the image during zooming.

The filter types available are

Filter Type	Category
IMPImpulse	Replicative
IMPBox	Filtered
IMPTriangle	Filtered
IMPQuadratic	Filtered
IMPMitchell	Filtered
IMPGaussian	Filtered

Refer to "Filter Functions" on page 192 for detailed information on the available filters.

blurFactor   Specifies a multiplier for the width of the filter. The default blur factor is 1.0. Higher factors increase the amount of blur present in the image.

The **impZoomRow()** function has the following parameters:

zoom   Pointer to a zoom operator structure obtained from a call to **impCreateZoom()**.

buffer   Caller-allocated buffer that is filled with the zoomed image row data. The buffer should be allocated to accommodated *dstXSize* * *numChannels* * **sizeof**(short) bytes.

row   The desired zoomed row. Rows are numbered from 0 through *dstYSize* minus 1. *clientData* is a pointer to client data. This pointer is passed to the *readRowFunc*. A typical use of this is for passing a pointer to a structure containing the **IMPImage** pointer and the channel number.

Filter Functions

The resampling filters available for zooming are summarized below. Note that the span of the filter (x range) is expressed in terms of the original image, not the zoomed image. Table B-3 lists the available filter functions.

Table B-3   Filter Functions

Filter Type	Function	Span
IMPImpulse	Not Applicable	
IMPBox	$f(x) = 0.0$	$x < -0.5$
	$f(x) = 1.0$	$-0.5 <= x < 0.5$
	$f(x) = 0.0$	$x >= 0.5$
IMPTriangle	$f(x) = 0.0$	$x < -1.0$
	$f(x) = 1.0+x$	$-1.0 <= x < 0.0$
	$f(x) = 1.0-x$	$0.0 <= x < 1.0$
	$f(x) = 0.0$	$x >= 1.0$
IMPQuadratic	$f(x) = 0.0$	$x < -1.5$
	$f(x) = 0.5*(x+1.5)^2$	$-1.5 <= x < -0.5$
	$f(x) = 0.75-x^2$	$-0.5 <= x < 0.5$
	$f(x) = 0.5*(x-1.5)^2$	$0.5 <= x < 1.5$
	$f(x) = 0.0$	$x >= 1.5$
IMPMitchell	$b = 1.0/3.0$	
	$c = 1.0/3.0$	
	$p0 = (6.0-2.0*b)/6.0$	
	$p2 = (-18.0+12.0*b+6.0*c)/6.0$	
	$p3 = (12.0-9.0*b-6.0*c)/6.0$	
	$q0 = (8.0*b+24.0*c)/6.0$	
	$q1 = (-12.0*b-48.0*c)/6.0$	

Table B-3 (continued)   Filter Functions

Filter Type	Function	Span
	q2 = (6.0*b+30.0*c)/6.0	
	q3 = (-b-6.0*c)/6.0	
	f(x) = 0.0	x < -2.0
	f(x) = q0-x*(q1-x*(q2-x*q3))	-2.0 <= x < -1.0
	f(x) = p0+x*x*(p2-x*p3)	-1.0 <= x < 0.0
	f(x) = p0+x*x*(p2+x*p3)	0.0 <= x < 1.0
	f(x) = q0+x*(q1+x*(q2+x*q3))	1.0 <= x < 2.0
	f(x) = 0.0	x >= 2.0
IMPGaussian	a(x) = 1.0/exp((1.5*x)^2)	
	b(x) = 1.0/exp(1.5^4)	
	f(x) = a(x)-b(x)	

Return Value:

The **impCreateZoom()** function returns a pointer to a zoom operator structure if execution was successful. NULL is returned and *IMPerrno* is set if an execution error has occurred.

The **impZoomRow()** function returns 0 if execution was successful. -1 is returned and *IMPerrno* is set if an execution error occurred.

Execution Error Codes:

The **impCreateZoom()** function fails with the following error:

- IMP_ERR_MEMALLOC

The **impZoomRow()** function fails with the following error:

- IMP_ERR_READROW

Note: The storage for the **IMPZoom** structure is allocated by the zoom operator creation function. This storage is deallocated by the **impDestroyZoom()** function. The caller should not explicitly reallocate or deallocate any storage related to the image structure. Furthermore, the fields of the **IMPZoom** structure are private and should not be modified by the caller.

See also:

libimp(3)

Printer Object Database (POD) File Formats

This appendix describes the file formats of the ASCII text files that compose the Printer Object Database. The default contents of these files and the interpretation is also described.

Printer Object Database (POD) File Formats

This appendix describes the file formats of the ASCII text files that compose the printer object database (POD).

The following major topics are discussed:

- "General Syntax" on page 199
- "Input Parsing Rules for libpod Files" on page 200
- "Printer Configuration File Format" on page 202
- "Printer Status File Format" on page 212
- "Printer Log File Format" on page 216

Overview

The printer object database (POD) contains information on the current configuration, status, and job history of a single printer. Each printer that is physically installed on a system must maintain its own POD. To maintain network-transparent, mediated access to the POD files, all interaction between the printer driver and POD files must be through the *libpod* API. For additional information, see "The libpod Library" in Chapter 5 and the libpod(3) reference page.

The initial set of POD files created by the printer driver developer and installed on the host server system must include the following:

- *<printer name>.config*:   a configuration file representing the printer's capabilities

- *<printer name>.stats*:   a status file indicating a typical printing state

- *<printer name>.log*:   an empty log file

The name of each POD file is formed from the printer name and the suffix *.config*, *.status*, or *.log*, respectively. Note that the printer name must be the same as the name of the printer model file. All POD files are copied from the template POD files and installed in the appropriate directory.

The information contained in each POD file is summarized below and explained in detail in subsequent sections.

Printer Configuration File

The printer configuration file (*<printer name>.config*) contains detailed information on the printer's capabilities. The file is created by the printer driver developer to characterize the printer's capabilities. The possible paper sizes, printer location, and available fonts are all specified in this file. Typically the information in the config file does not change over time. Printer filters and drivers treat it as a read-only file. The printer install tools may modify the file at printer installation time to enter site-specific data, such as printer location, or note the presence of optional equipment, such as a duplex option or an envelope feeder.

Printer Status File

The printer status file (*<printer name>.status*) contains information about the current operational status of the printer. The information in the file indicates whether the printer is busy, what type of printing media is installed, and so on. The contents of this file change during every print job. The driver developer provides an initial copy of the status file, but it is the job of the printer driver to update the file. Printer filter programs normally treat this file as if it were a read-only file.

Printer Log File

The printer log file (*<printer name>.log*) contains the print job history for the printer. Information for old jobs as well as the current print job is maintained. Typically, printer filters and drivers append information to the log file, while general applications treat the file as if it were a read-only file.

General Syntax

Character Set

```
<space>:        0x09, 0x20 (<sp>, <ht>)
<null>:         0x00-0x08, 0x0B, 0x0C, 0x0E-0x1F, 0x7F-0xFF
<endline>:      0x0A, 0x0D (<nl>, <cr>)
<separator>:    0x7C ('|')
<plainchar>:    0x21-0x7B, 0x7D, 0x7E
<ddigit>:       0x30-0x39 ('0'-'9')
<hdigit>:       <ddigit>, 0x41-0x46, 0x61-0x66
                ('0'-'9', 'A'-'F', 'a'-'f')
<sign>:         0x2B, 0x2D ('+', '-')
<point>:        0x2E ('.')
```

Field Format

```
<white>:        <space> [<space>...]
<word>:         <plainchar> [<plainchar>...]
<keyword>:      <word> with a specific sequence of <plainchar>
<keyfield>:     [<white>] <keyword> [[<white> <keyword>]...]
                [<white>]
<string>:       [<white>] <word> [[<white> <word>]...]
                [<white>]
<int>:          <ddigit> [<ddigit>...]
<hbyte>:        ["0x" | "0X"] [<hdigit>] <hdigit>
<float>:        [<sign>] <int> [<point> [<int>]]
                or
                [<sign>] <point> <int>
<array>:        <string> [[<separator> <string>]...]
```

Input Parsing Rules for libpod Files

The following rules apply when a POD file is parsed by the *libpod* API:

- All `<null>` characters are ignored. Their use is discouraged to avoid errors caused by nonprinting characters appearing in the POD files.

- All input lines are truncated to PD_STR_MAX-1, not counting `<null>` characters and `<endline>`, which are removed on input. The value of PD_STR_MAX is defined in the header file */usr/include/pod.h.*

- All occurrences of `<white>` sequences are reduced to a single `<sp>` character. Any `<white>` at the beginning or end of a field is removed.

- There are no quoted strings. Quotation marks are treated like any other characters and cannot be used to force additional `<white>` into a field.

- All fields are checked for correct syntax based on entry type. Failure to provide information in the correct format results in improper parsing.

- When scanning for `<int>` or `<float>` numbers within a field, all characters that are not valid within an `<int>` or `<float>` are treated as `<white>` (in the case of an `<int>`, `<sign>` and `<point>` are treated as `<white>`). This allows characters to be inserted to improve readability. For example, the following are equivalent if two `<int>` items are expected:

  - 300 300

  - 300 x 300

  - 300 by 300

  - 300,300

- Entries containing no characters other than `<white>` before the first `<separator>` or `<endline>` are treated as null entries and discarded without error. These lines may be used as comments by placing a `<separator>` before any other information.

- Blank lines are ignored and may be inserted to improve readability.

- `<keyfield>` matching is done in a case-independent manner.

- Fields designed to be human-readable are not modified, except to remove `<null>` and excess `<white>`. Case and all `<plainchar>` sequences are preserved.

- A `<keyfield>` may require a long list of items (for example, *Available Fonts*). To improve readability and avoid the risk of input line buffer overflow (see the second bullet item above), a `<keyfield>` may be repeated.

  For example, a list of fifty *Available Fonts* items may be broken into two *Available Fonts* entries with 25 items each. The overall number of items that can be specified in a list is limited only by available system memory resources.

- There is no required entry order. The `<keyfield>` entries may appear in any order within a POD file.

- Default values are assumed for certain fields if values are not specified. The values of these defaults should not be relied upon and may change in future releases.

Printer Configuration File Format

This section describes the format of the printer configuration file. The configuration file is installed by the printer install tools with the name *<printer name>.config*.

General Format

The format for an entry in the configuration file is

```
<keyfield> <separator> [<infofield>] <endline>
```

where

`<keyfield>`	is one of the reserved fields described in the "Key Field" column of Table C-1.
`<separator>`	is the "separator" character defined in "Character Set" on page 199.
`<infofield>`	is one or more of the options specified in the "Info Field Type" column of Table C-1.
`<endline>`	is one of the "endline" characters defined in "Character Set" on page 199.

Config File Options

All entries in the config file are optional. Entries that are not provided or that have no `<infofield>` specified are assigned default values. However, since printer capabilities differ, it is strongly recommended that no entries be omitted. The defaults should not be relied on, as they may change in future releases. Table C-1 lists the config file options in alphabetical order. A detailed explanation of each option follows the table.

Table C-1   Config File Options

Key Field	Info Field Type	Default
Active Status Path	<word>	See "Active Status Path" on page 204.
Available Fonts	<array>	(0 elements)
Black Substitute	<keyword>	No
Color Adjustment	<array>	(0 elements)

Table C-1 (continued)   Config File Options

Key Field	Info Field Type	Default
Cost per Page	<float>	0.00
Default CA	<int>	0
Default IS	<int> [,gamma=<float>]	0, gamma=-1.0
Default MT	<int>	0
Default QM	<int>	0
Driver Path	<word>	See "Driver Path" on page 207.
Error Retry Wait	<int>	10 seconds
Input Source	<array>	(0 elements)
Location Code	<keyword>	None
Manual Capable	<keyword>	No
Maximum Addr	<Maximum Addrint> <int>	See "Minimum Addr" on page 208.
Maximum Print Area	<float> <float>	See "Maximum Print Area" on page 208.
Media Standard	<keyword>	American
Media Type	<array>	(0 elements)
Media Wait	<int>	300 seconds
Minimum Addr	<int> <int>	See "Minimum Print Area" on page 209.
Minimum Print Area	<float> <float>	See "Minimum Print Area" on page 209.
Number of Colors	<int> [<int>]	1 1
Physical Location	<string>	Unknown
Port Path	<word>	/dev/null
Printer Class	<keyword>	Dumb
Printer Model	<string>	Unknown

Table C-1 (continued)   Config File Options

Key Field	Info Field Type	Default
Printer Options	<string>	(empty string)
Quality Modes	<array>	(0 elements)
Resolution	<int> <int>	300 dpi 300 dpi
Size Table Entry	<sizeentry>	See "Size Table Entry" on page 210.
Status Update Wait	<int>	300 seconds
Technology	<string>	Unknown
Time per Page	<int> [<int>]	0 0

Active Status Path

Active Status Path is the full pathname of the POD status file. The value of this entry is not used by the *libpod* API. The value of this entry is always set by the API to *PDpod_path/<printer name>.status.* Refer to the libpod(3) reference page for additional information.

Available Fonts

The *Available Fonts* option contains a list of font names representing the fonts available on the printer. For printers with built-in PostScript interpreters, this list should include only those fonts built into the printer (typically a set of 35 standard fonts). The default value is 0 elements.

For raster printers, the PostScript interpretation is performed on the printer host machine. Thus, the fonts listed for these printers should correspond to the names of the font outline files installed on the printer host. There are two methods for specifying the font names: the names can be listed individually or a full path to the directory where the outline fonts are stored can be specified. The two methods can be mixed. When a path is specified, the names of the files in that directory are assumed to be the names of the fonts. To exclude filenames from the directory, specify the names of the files to be excluded with a leading "!". The filenames to be excluded must appear on the same line as the directory containing the filename to be excluded. The following is a valid *Available Fonts* list:

```
NewYearRoman | /usr/lib/DPS/outline/base | !fonts.dir
```

This entry indicates that the only font available on the printer is NewYearRoman, and all file names are in the directory */usr/lib/DPS/outline/base* with the exception of *fonts.dir*. Note that font names must not contain any white space.

Starting with Impressario 2.0, a new logical AND operator, **&**, is available for font lists. It allows the list of fonts on the printer to be logically AND'ed with the list of fonts on the print server, generating a list of fonts that are common to both. The common list is needed for the text2ps(1) filter, which uses the font metric information from the print server fonts to determine where line and page breaks should be made. The AND operator is intended for use by printer drivers that send PostScript directly to the printer.

The entry below says that NewYearRoman is available on the printer, and if it is also available in */usr/lib/DPS/outline/base*, it should be on the list of available fonts when a printer driver is installed:

```
| NewYearRoman | &/usr/lib/DPS/outline/base
```

For another example, see the file

/usr/lib/print/data/lexmarkoptra_model.config

Black Substitute

The *Black Substitute* option is either *yes*, indicating that the printer should by default substitute true black for composite black, or *no*, indicating that it should not. The default value is *no*.

Color Adjustment

The *Color Adjustment* option is a list of color adjustment methods available for the printer. The color adjustment methods perform color correction between the current input source and the printer. An example entry is

```
None | Fix Blue | Gamma Correct
```

The default value is 0 elements.

Starting with Impressario 2.0, International Color Consortium (ICC) color profiles and various filters, such as *cocostiff*, are used to perform color management. Color correction is applied as a filter before the driver is invoked (FTR rules have been added so *fileconvert* will invoke the correct filters based on the file type). The Color Adjustment entry in the POD file is currently not used. See Appendix G for more information on color management in Impressario.

Cost per Page

The *Cost per Page* option is the cost per printed page in local currency. For example, 0.50 for 50 cents per page. The default value is 0.00.

Default CA

The *Default CA* option is the index to the *Color Adjustment* list indicating the default adjustment method. This index is based at one rather than zero. Thus, the first method in the list is at position 1, the second at 2, and so on. If there are no adjustment methods specified, this entry should either be left empty or set to 0. The default value is 0.

Default IS

Default IS is the index to the *Input Source* list indicating the default input source. This index is based at one rather than zero. Thus, the first source in the list is at position 1, the second at 2, and so on. If there are no input sources specified, this entry should either be left empty or set to 0.

When used for printer color correction, this entry can also be used to specify the default device's gamma correction value. The gamma value is specified after the default input source index, as in the following example:

```
1, gamma = 1.0
```

Default MT

Default MT is the index to the *Media Type* list indicating the default media. This index is based at one rather than zero. Thus, the first media in the list is at position 1, the second at 2, and so on. If there are no media types listed, this entry should either be left empty or set to 0. The default value is 0.

Default QM

Default QM is the index to the *Quality Modes* list indicating the default quality mode. This index is based at one rather than zero. Thus, the first quality mode in the list is at position 1, the second at 2, and so on. If there are no quality modes specified, this entry should either be left empty or set to 0. The default value is 0.

Driver Path

Driver Path is the full pathname of the installed printer driver. The default is the full pathname of the POD config file, with the suffix *.config* removed.

Error Retry Wait

Error Retry Wait refers to the number of seconds to wait after an error has occurred before attempting to resume printing. The default is 10 seconds.

Input Source

Input Source is a list of printer input sources. The primary use of this entry is to list the image source devices that have been characterized for printer color correction. A common input device is the monitor. An example entry is *Sony 16" Monitor*.

Location Code

The *Location Code* option is used to supply a site-specific keyword that identifies the printer's physical location. For example, 3U-924. There is no default value. This code should be machine-readable and -sortable for use by a location-querying browser.

Manual Capable

Manual Capable is either *yes*, indicating that the printer is capable of being manually fed, or *no*, indicating that it is not. The default value is *no*. This value should be set to *yes* only if the printer supports the -**m** option (see "The Filter/Driver Specification and psrip" in Chapter 2).

Maximum Addr

Maximum Addr is the maximum printable area dimensions expressed in dots. The default values for this entry assume an A-size page (8.5 by 11.0 inches) in portrait orientation with 0.5-inch margins. At 300 dpi, this gives a printable area of 2250 by 3000 dots. The minimum and maximum values are identical in the default case. If a *Page Size Table* has been specified, the values for this entry are derived from it. Also see "Minimum Addr."

Maximum Print Area

Maximum Print Area is the maximum printable area dimensions expressed in inches. The default values for this entry assume an A-size page (8.5 by 11.0 inches) in portrait orientation with 0.5-inch margins. This gives a printable area of 7.5 by 10.0 inches. The minimum and maximum values are identical in the default case. If a *Page Size Table* has been specified, the values for this entry are derived from it. Also see "Minimum Print Area."

Media Standard

Media Standard indicates the paper measurement standard. Keywords are **American** and **Metric**. The default value is **American**.

Media Type

Media Type is an array that contains the output media types supported by the printer. Typical items are **Bond Paper** and **Transparency Film**. The default array has *0* elements.

Media Wait

Media Wait is the number of seconds to wait for manual feed or print media changes before the default media source is used. The default is 300 seconds.

Minimum Addr

Minimum Addr is the minimum printable area dimensions expressed in dots. The default values for this entry assumes an A-size page (8.5 by 11.0 inches) in portrait orientation with 0.5-inch margins. At 300 dpi, this gives a printable area of 2250 by 3000 dots. The minimum and maximum values are identical in the default case. If a *Page Size Table* has been specified, the values for this entry are derived from it. Also see "Maximum Addr."

Minimum Print Area

Minimum Print Area is the minimum printable area dimensions expressed in inches. The default values for this entry assume an A-size page (8.5 by 11.0 inches) in portrait orientation with 0.5-inch margins. This gives a printable area of 7.5 by 10.0 inches. The minimum and maximum values are identical in the default case. If a *Page Size Table* has been specified, the values for this entry are derived from it. Also see "Maximum Print Area."

Number of Colors

Number of Colors is the minimum or, optionally, maximum number of colors that are available on the printer. If the maximum number of colors is not provided, it is assumed to be the same as the minimum. A monochrome printer or printer ribbon provides one color. A CMY printer or ribbon provides three colors.

Note that this field should contain only the number of colors available on the printer. The *colorspace, depth*, and *data format* are provided in the *Number of Colors* entry in the status file. The default value is 1.

Physical Location

Physical Location is the human-readable description of the printer's physical location. For example, Bldg. 3 Upper, Room 924. The default value is **Unknown**.

Port Path

Port Path is the full pathname of the I/O port to which the printer is physically connected. For example, */dev/plp* for a parallel printer, */dev/ttyd2* for a serial printer, and */dev/scsi/sc0d6l0* for a SCSI printer. The default value is */dev/null*.

Printer Class

The *Printer Class* entry is used to specify the class of printer being used. Available values are **Dumb**, **Raster**, **ColorRaster**, **MonoPostScript**, **ColorPostScript**, and **Plotter**. The following printer classes are obsolete and should not be used for new development: **DumbColor**, **PostScript**, and **Color**. The default value is **Dumb**.

Printer Model

The *Printer Model* entry is a keyword that is the manufacturer's description of the printer. For example, Tektronix Phaser II SX or Apple LaserWriter II NTX. The default value is **Unknown**.

Printer Options

Printer Options describe the standard installed printer optional equipment. For example, 8 megabytes RAM. The default is an empty string.

Quality Modes

Quality Modes is a list of output quality modes available on the printer. For example, **draft** and **letter**. The default value is 0 elements.

Resolution

Resolution is the horizontal or vertical printer resolution in dots per inch (dpi). For printers that allow multiple resolutions, the status file *Printer Options* entry should be parsed for the *CurrentRes* keyword. This keyword indicates the current printer resolution. If the keyword is not found, the config file *Resolution* entry should be used. The default values are both 300 dots per inch.

Size Table Entry

The *Size Table Entry* `<infofield>` has the format `<sizeentry>` defined as

```
<sizeentry>: <keyword> <int> <int> <float> <float> <float>
             <float> [<hbyte> [<hbyte>]]
```

Size Table Entry describes a particular media size that is supported by the printer. Typically, there is one *Size Table Entry* per supported media size (for example, an entry each for A size and B size), although it is acceptable to have multiple entries for a paper size if multiple resolutions or ribbons are supported. The media size entry has seven required fields and two optional fields. All fields are separated by white space.

The first required field contains the *Media Size* (for example, A). The list of possible media sizes can be found in the file */usr/include/pod.h*. The media size keyword is simply the media size listed in *pod.h* with the *PD_SIZE_* prefix removed. The size names listed in *pod.h* with the suffix *_LAND* indicate landscape orientation and should not be used as media size keywords. Media with landscape orientation is indicated by the width and height fields of the *size* table entry.

The next two fields are the media-imageable width and height expressed in dots. Typically, the imageable dimensions are derived by subtracting the margins from the total media size and converting the result to dots.

The next two fields are the overall media width and height expressed in inches.

The last two required fields are the left and top margins expressed in inches.

The first optional field specifies the printing raster direction. Refer to *pod.h* for the values that may be specified in this field.

The second optional field is the media validation mask. This mask can be used to differentiate among media entries that have the same media name but differ in other respects (for example, resolution). The field is a bit mask and so, to fully differentiate among similar entries, the values must be powers of two. Refer to the PDReadInfo(3) reference page for more information on the use of this field.

A default *Size Table Entry* is always added to the end of the table when it is read by *libpod*. This default entry is

```
A 2250 3000 8.500 11.000 0.500 0.500 0x00 0xFF
```

Status Update Wait

Status Update Wait is the number of seconds to wait between updates of the POD status file when no error has occurred. The default is 300 seconds.

Technology

The *Technology* option indicates the type of printing technology used in the printer. For example, **ink jet**, **wax transfer**, **dye sublimation**, and **color laser**. The default value is **Unknown**.

Time per Page

Time per Page is the average and, optionally, maximum time to print a page, in seconds. If the maximum time is not provided, it is assumed to be the same as the average time. The default values are both 0.

Printer Status File Format

This section describes the format of the printer status file. The status file is installed by the printer install tools with the name *<printer name>.status*.

General Format

The format for an entry in the printer status file is

`<keyfield> <separator> [<infofield>] <endline>`

where

`<infofield>`   is the parameter is specified by Table C-2.

`<separator>`   is the separator defined in "Character Set" on page 199.

`<endline>`   is one of the endline characters defined in "Character Set" on page 199.

Printer Status File Entries

All entries in the status file are optional. Entries that are not provided or that have no <infofield> are assigned default values. However, since the status file is the only means to indicate printer status to the user, it is strongly suggested that a complete status file be provided by the developer and that the printer driver update the status file to reflect the printer's current status.

Table C-2 lists the printer status file entries.

Table C-2   Printer Status File Entries

Key Field	Info Field Type	Default
Operational Status	<keyword>	Idle
Media Size	<keyword> [Land]	A
Media Type	<keyword>	Paper
Number of Colors	<dataentry>	1 k 1 chunky
Printer Options	<string>	(empty string)
Validation Mask	<hbyte>	0
Error	<msgentry>	(no message)
Warning	<msgentry>	(no message)
Information	<msgentry>	(no message)

Operational Status

Operational Status is the keyword specifying the current operational status of the printer. The possible values are **Idle**, **Busy**, **Faulted**, and **Unavailable**.

The status **Faulted** indicates that there is a problem with the printer but not with communication to the printer. The **Unavailable** designation is similar to the **Faulted** state but indicates that communication could not be established with the printer. The default value is **Idle**.

Media Size

Media Size is the keyword indicating the currently loaded media size. The media size keywords are listed in the file *pod.h.* The keyword is the size name listed with the *PD_SIZE_* prefix removed. The size names listed in *pod.h* with the suffix *_LAND* indicate landscape orientation and are specified in the entry by the keyword *Land.*

Media Type

Media Type is the keyword indicating the currently loaded media type. The value is **Paper**, **Transparency**, **Other**, or **Unknown**. The default is **Paper**.

Number of Colors

This field specifies not only the number of output colors but the colorspace, depth, and organization of the output data. There is one required field and three optional fields. For proper operation of printing filters, it is strongly recommended that the optional fields be specified.

The *Number of Colors* `<infofield>` has the format `<dataentry>` defined as

`<int> [<keyword> <int> <keyword>]`

Arguments:

| `<int>` | Required. Specifies the number of output colors the printer can currently print. If only this field is present, the following defaults apply: |

# of colors	colorspace	depth	organization
1	k	1	chunky
3	cmy	1	chunky
4	cmyk	1	chunky

`<keyword>`	Optional. This field specifies the output colorspace and is either **k**, **rgb**, **cmy**, **ymc**, **cmyk**, **ymck**, **w**, or **kcmy**.
`<int>`	Optional. This field specifies the number of bits per color component and may be 1, 4, or 8.
`<keyword>`	Optional. This field specifies the data organization of the output data and is either **chunky** or **planar**.

An example output specification is

```
3 rgb 4 planar
```

This specifies a three-color RGB output with four bits per component (12 bits total) and a planar data organization. Refer to the libstiff(3) reference page for additional information on raster data output formats.

Printer Options

This field is used to describe the currently available optional equipment or configurations. The field is also used to indicate the current printer resolution for printers that allow multiple output resolutions. To indicate the current resolution, the string

```
CurrentRes = <int> x <int>
```

is specified. The first integer is the horizontal resolution in dots per inch, and the second integer is the vertical resolution in the same units. The current resolution values are used by printing filters such as the PostScript interpreter *psrip* to calculate margins for printers whose resolutions can change, and it is very important that printer drivers update this information field to ensure proper rendering.

Validation Mask

This field can be used to differentiate among media entries that have the same media name but differ in other respects (for example, resolution). The field is a bit mask and so, to fully differentiate among similar entries, the values must be powers of two. Refer to the PDReadStatus(3) reference page for more information on the use of this field.

Error, Warning, and Information Options

Each Error, Warning, and Information message `<infofield>` has the format `<msgentry>` defined as

`<hbyte> [<hbyte> [<hbyte>]] <separator> <string>`

These three entries indicate messages written by the printer driver to provide information to the printer user regarding the state of the printer. The three hex bytes provide a message code. The available message codes are listed in *pod.h* (see PD_ERROR_*). The low-order three bytes of the codes listed in *pod.h* are the codes specified in this field. The high-order byte of the code is implied by the first field (for example., *Information* = 00, *Warning* = 01, *Error* = 02). The last field is a string providing the text for the message. There can be up to PD_MESSAGE_MAX (see *pod.h*) message entries in a status file. An example of a complete message entry is

`Information | 01 00 00 | version: driver = 00.00`

See the **PDMakeMessage()** routine for the best method of construction these messages. For the sake of internationalization, it is strongly recommended that you do not customize messages.

Printer Log File Format

The log file is not currently implemented. You should supply an empty file in */usr/spool/lp/pod* with the name *<printer name>.log,* and should not write to the log file.

Transition Notes

This appendix explains how Impressario application developers and filter/driver developers can take advantage of the new features in Impressario 2.0.

Transition Notes

This appendix explains how application and printer driver developers can take advantage of the new features in Impressario 2.*n*.

The following major topics are discussed:

- "Notes for Application Developers" on page 220
- "Notes for Printer Driver Developers" on page 220
- "General Changes in IRIX 6.2" on page 226

Notes for Application Developers

Changes that application developers should be aware of are

- PrintPanel(1) only runs printer graphical options panels that are in the */var/spool/lp/gui_interface/ELF* directory. Because IRIX 6.2 does not run COFF executables, graphical options panels in the COFF-related */var/spool/lp/gui_interface* directory are ignored. This change can be seen in *PrintOptionPanel.c* in */usr/impressario/src/libprintui*.

- libimp(3) now supports International Color Consortium (ICC) color profiles embedded in SGI image files. It is available as a MIPS2 32-bit, a MIPS3 N32 (high performance 32-bit), or a MIPS3 64-bit library.

- libprintui(3X) is available as a MIPS2 32-bit or MIPS3 N32 library.

See the *c_dev* release notes for information about 32-bit, n32 and 64-bit compilations.

Notes for Printer Driver Developers

Changes to the PostScript Interpreter (psrip)

Note: Adobe Systems, Inc. imposes strict limitations on the use of the Configurable PostScript Interpreter by developers. Please refer to Appendix H, "Using the Adobe Configurable PostScript Interpreter," for details.

The PostScript interpreter, psrip, has been upgraded to an Adobe Level II Configurable PostScript Interpreter (CPSI).[1] Two significant improvements are its ability to set the gamma correction for individual color channels, and its ability to invoke psrip as a band device as well as a frame device. (You can also let it choose between them.) A band device renders the output raster image as a series of sequential bands rather than as a complete entity. It thus usually requires much less memory than a frame device. (A minimum band size of 5 megabytes is recommended.) See the psrip(1) reference pages for all the new features.

---

[1] You can find the Adobe PostScript version number of psrip by using the command
`/usr/lib/print/psrip -x`

psrip is the only executable in Impressario 2.*n* that is licensed. It uses a FLEXlm node-locked license, stored in the file */var/flexlm/license.dat,* and returns an error if the license is not available. You can use the License Manager in the System toolchest menu to update software licenses. For more information on obtaining a license, see the file */usr/lib/print/data/psrip_expired_msg.* Additional information on FLEXlm can be found in the online manuals *FLEXlm End User Manual* and *FLEXlm Programming Guide.*

psrip always runs as set user ID lp to guarantee that its files have the same ownership.This is necessary because psrip creates temporary and permanent files in */var/spool/lp/psparams* and needs to be able to modify them.

Halftone screens are found in */usr/lib/print/data/screens.* Three screens are available:

- Impr1_2_Default.   This is the default screen used in Impresario 1.2. It provides a clustered-style threshold array.

- Impr2_0_Default.   This is the Impressario 2.0 default screen. Because all default screen values are defined within */usr/lib/print/data/psrip/startup.ps* or psrip, the file itself is currently empty. The file is here in case the default changes in the future.

- Impr2_0_Default_UA.   Similar to the Impr2_0_Default screen, its colors (C,M,Y,(K)) are un-aligned: They are not placed directly on top of each other. This results in colors that are more saturated at the expense of having some grays and blacks with a purple cast. It is recommended for image files like TIFF and GIF.

The psrip -**I** option can be used to select between these screens. See the model file */usr/impressario/src/models/laserjetPJL_model* for examples.

Fonts are in the directory*/usr/lib/DPS.* psrip uses the */usr/lib/DPS/AdobeFonts.upr* file to determine the available fonts. The makepsres(1) function, which is automatically run (as an **exitop**) when the user quits the Software Manager after installing the *dps_eoe.sw.dps* sub-system, builds */usr/lib/DPS/AdobeFonts.upr* based on the installed fonts.

The directory */usr/lib/print/data/psrip* contains other resources used by psrip including Color Rendering Dictionaries (CRDs) and the default *startup.ps* file.

There are some other changes in psrip:

- The CHUNKY output formats **oldk**, **oldcmy**, and **oldcmyk,** provided for compatibility with previous releases of Impressario, have been removed.

- psrip now ignores the -**T** option.

- The psrip command-line option **rotate** is now limited to rotating an image in 90 degree increments.

Changes Affecting Model Files

For Impressario 2.*n*, there is an updated sample model file for HP LaserJet printers, */usr/impressario/src/models/laserjetPJL_model*. The Impressario 1.2 version of the *laserjet_model lp* model file is still available for comparison and is found in the directory */usr/impressario/src/models*. There are several major changes in the updated file:

- Many of the printer drivers now write to standard output instead of directly to the parallel port. This allows the driver output to be piped to a device-specific driver such as *phandler*, which drives printers from the parallel port, or *nethandler*, which drives printers from network adapters such as the HP JetDirect network adapter. See the file */usr/impressario/src/models/laserjetPJL_model* for examples.

- Although lptops is still available, there is a new ASCII-text-to-PostScript filter, text2ps.

  Note: All Impressario model files now use text2ps to convert ASCII text files to PostScript and your model files should also.

  Tip: Unlike lptops, text2ps cannot print pages in reverse order. To reverse pages with text2ps, use the psselect filter. (The file type rules used by *fileconvert* have been updated to use psselect to do this.)

- Starting with IRIX 6.2, the *man* -**t** command now invokes */usr/lib/print/manprint* which calls *lp* with the -**o**"-**manpage"** option if Impressario is installed. Impressario model files have been updated to support this. You should add similar support to your reference pages.

- As mentioned in the previous section, psrip uses a FLEXlm license. Model files should check for a valid license and disable the print spooler if one is not found. See the */usr/impressario/src/models/laserjetPJL_model* file for an example.

Other changes of interest in Impressario 2.*n* are as follows:

• text2pcl has been updated to support A3 and B paper sizes.

• A bug that prevented the OPTION string in the model files from being parsed has been fixed. It is now possible for a string like

```
OPTIONS=numcolors=3 gamma=1.0 default_profile=hp500c06.pf
```

to set the *numcolors, gamma,* and *default_profile* variables when the Printer Manager is used to install the model file into */var/spool/lp/interface*.

• When using psrip, you can set the gamma value for each color channel. See the */var/spool/lp/model/deskjetII_model* model file for an example.

Finally, releases of IRIX after version 6.2 will not support CTR (Compiled Type Rule) database files. Your model files may be checking for this file using code like this:

```
if [ ! -r /usr/lib/filetype/workspace.ctr ]; then
    disable -r"WorkSpace filetype database not built. Type 'su; cd
/usr/lib/filetype; make'" \
    $printer 1>>$logfile 2>&1
    exit 2
fi
```

You should remove or comment out the above block of code. Future versions of *fileconvert* will map the old filename, *workspace.ctr*, to the new name, *desktop.otr*, to maintain backward compatibility. Even so, the file */usr/lib/filetype/workspace.ctr* may not exist, so you should still remove the above code.

Changes Affecting POD Data Files

Starting with Impressario 2.0, a new logical AND operator, **&**, is available for font lists. Now the list of fonts on the printer can be logically AND'ed with the list of fonts on the print server, generating a list of common fonts that the graphical options panel can present to the user. The common list is important because the text2ps filter uses the font metric information of the print server fonts to determine where line breaks and page breaks should be made. The AND operator is intended for printer drivers that send PostScript directly to the printer. See Appendix C, "Printer Object Database (POD) File Formats," for an example.

Changes Affecting Printer Drivers

Some changes that affect printer drivers are

- The laserjet driver has been updated and renamed *laserjetPJL*. The source code is available in the */usr/impressario/src/drivers/laserjetPJL* directory.

- Printer drivers that use the parallel port now check only the first eight characters of the output device, instead of looking for it to be */dev/plp*. This allows them to support */dev/plp1*, */dev/plp11*, and so on. Developers should update their drivers to behave in a similar manner.

- Most printer drivers that directly controlled the parallel port now send output to standard output. The output is then piped to *phandler* or *nethandler* in the model file.

  Note: A general purpose driver for the serial port does not exist.

- *phandler* has been updated. The source code is available in the directory */usr/impressario/src/drivers/phandler*.

Another change may be necessary for drivers using *nethandler*. To avoid network printer timeouts, printer drivers that are meant to be used with *nethandler* should not send data to standard output until they read something from standard input. By waiting, the drivers can avoid initializing the printer prematurely because *nethandler* won't open a socket connection to the printer until it receives data from the driver via standard input.

Here is the underlying problem. When the printer driver is invoked from a model file,

```
image2ps image_file | psrip | printer_driver | nethandler
```

a delay is introduced as the programs upstream from the printer driver process the data. If the image being processed is complex, the printer driver may not see data on standard input for some time. If the printer driver sends initialization commands to the printer immediately, *nethandler* will open a socket connection to the printer to deliver them. Most network printers then start a timeout sequence and close the socket if they do not see any more data before the timeout expires (typically 90 seconds to 2 minutes). If the processing upstream from the printer driver takes longer than the timeout period, the socket connection will be closed by the printer and *nethandler* will return an error. Recovery from this is difficult because trying again will probably have the same result.

Current printer drivers probably perform this sequence of events (which risks a timeout):

```
parse command line
send commands to printer to initialize it
while (data to read from standard input) do
   read data from input
   process data and format for printer
   output data to printer
done
cleanup
```

The timeout problem can be addressed by modifying the printer drivers so:

```
parse command line
while (data to read from standard input) do
   read data from input
   process data and format for printer
   (if never initialized) send commands to printer to initialize it
   output data to printer
done
cleanup
```

Changes to the Graphical Options Panel

Tip: When a print client is configured, it copies the Graphical Printer Options Panel executable from the print server to */var/spool/lp/gui_interface/ELF*. By default, IRIX 6.2 generates o32 MIPS2 ELF binaries. MIPS2 executables will not run on IRIX 5.3 systems with a MIPS R3000 processor, the configuration of many Indigo workstations. (IRIX 6.2 is not supported on R3000 systems.) Compile the Graphical Printer Options Panel under IRIX 5.3 if you wish to support IRIX 5.3 systems running on a R3000. The subsystem *impr_dev.sw5_3* contains all of the Impressario 2.0 libraries compiled under IRIX 5.3. The libraries are installed in */usr/impressario/53libs* and can be copied to your 5.3 development system.

All of the Graphical Printer Options Panel executables supplied in Impressario 2.0 were compiled and linked under IRIX 5.3.

The Graphical Printer Options Panel now has a context-sensitive help panel at the top of the window. This gives a short description of any option pointed to with the mouse. See the sample source code in */usr/impressario/src/gui_models/laserjetPJL*.

General Changes in IRIX 6.2

Because IRIX 6.2 does not support COFF executables (IRIX 6.2 compilers cannot even generate COFF executables) developers need to ship ELF executables. They need to ship COFF executables only if they wish to support IRIX 4.0.5 systems. IRIX 6.2 includes the utility */usr/sbin/coffcheck,* which warns users about the presence of COFF executables. *coffcheck* ignores a COFF executable in the */var/spool/lp/gui_model* or */var/spool/lp/gui_interface* directory if there is an ELF executable of the same name in */var/spool/lp/gui_model/ELF* or */var/spool/lp/gui_interface/ELF,* respectively. This arrangement allows a 6.2 print server to support a 4.0.5 print client, because a 4.0.5 client will request the COFF version of the Graphical Printer Options file.

Other features that are new with IRIX 6.2 are listed below:

- The fonts found in the Impressario 1.2 subsystem *impr_fonts.sw.adobe22* are bundled with the base operating system. They have been removed from Impressario.

- The Printer Manager now lists SCSI devices that identify themselves as CPUs, such as certain dye-sublimation printers.

- When a printer is being added, the Add Printers menu of the Printer Manager displays a connection type, such as SCSI, only if there is a driver installed that supports that connection type.

Scanner Driver Architecture

This appendix discusses scanner driver architecture. It provides a detailed analysis and discussion of the template scanner driver.

Scanner Driver Architecture

This appendix discusses scanner driver architecture. It provides a detailed analysis and discussion of the template scanner driver.

The following major topics are discussed:

- "Overview" on page 230
- "Driver Structure" on page 230
- "Type Conversion Macros" on page 238
- "Scanner Functions" on page 232
- "Queues and Multi-Threaded Scanner Drivers" on page 241

Overview

Scanner drivers are programs that are executed by applications that link with *libscan.a* and call SCOpen(3). They accept commands from the application via an input pipe and return results via an output pipe.

All scanner drivers must implement the basic set of commands so that any application using the *libscan* interface can have access to the functionality offered by the scanner. Many library routines are provided for scanner driver developers to implement functionality in software that may not be implemented in hardware for some scanners. The support routines for writing scanner drivers can be found in *libscan.a*.

Driver Structure

A scanner driver consists of a number of functions that implement the set of commands required to drive the scanner. In the main routine, a table of these functions, with the position of each function in the table corresponding to its SCN **#define** in *scanipc.h*, is passed to **SCDriverSetCallbacks()**, then **SCDriverMainLoop()** is called.

SCDriverMainLoop() waits for input from the application and calls the function in the table corresponding to each command received. Each function has the following prototype:

```
void scanfunc(int cmd, SCARG *arg, SCRES *res);
```

Arguments:

cmd   Contains the SCN **#define** of the command to be executed.

arg   Is the argument to this scanning function. **SCARG** is defined in *scandrv.h* as follows:

```
typedef struct tag_scarg {
    void *data;
    int len;
} SCARG;
```

arg->data points to the arguments transferred from the application; the meaning of *arg->data* depends upon the *cmd* (see below).

arg->len encodes the byte length of *arg->data*.

res   Is the result of this scanning function. **SCRES** is defined in *scandrv.h* as
               follows:

```
typedef struct tag_scres {
    void *data;
    int len;
    void *freeparam;
    void (*free)(void *param, void *data);
    int errno;
    char *errmsg;
} SCRES;
```

res->data should be set to point to the data to be returned to the
application as a result of *cmd.*

res->len is the byte length of *res->data.*

res->free is a pointer to a function that is called if it is nonzero after
res->data has been transferred to the application. The function is called
with *res->freeparam* as its first argument and *res->data* as its second
argument.

res->errno should be set to one of the SCE **#define**s in
/usr/include/scanner.h or one of the *errno* values from
/usr/include/sys/errno.h if an error occurs during the execution of *cmd.* If
res->errno is nonzero, the *libscan* function being executed on the
application side returns an error status, and *SCerrno* is set to the value
of *res->errno.*

res->errmsg is the error message pointer. If *res->errno* is set to
SCEDRVMSG, then *res->errmsg* should point to a driver-specific error
message.

Before a scanning function is called, the entire *res* structure is zeroed. Scanning functions
are allowed to assume that any member of the *res* structure not explicitly set remains set
to 0.

Scanner Functions

Required Scanner Functions

All scanner drivers must implement the functions listed in Table E-1.

Table E-1   Scanner Driver Functions

Function	Description
SCN_INITOK()	Checks for successful scanner driver initialization
SCN_PAGESIZE()	Returns the size of the scan area that is supported by the scanner
SCN_MINMAXRES()	Returns the smallest and largest horizontal and vertical resolution
SCN_NRES()	Returns the number of resolution pairs supported in hardware
SCN_RES()	Returns floating-point numbers representing the supported hardware resolutions
SCN_NTYPES()	Returns the number of data types supported by the driver
SCN_TYPES()	Returns an array of **SCDATATYPE** objects, one for each of the types supported by the driver
SCN_FEEDERGETFLAGS()	Gets the document feeder flags
SCN_FEEDERSETFLAGS()	Sets the document feeder flags
SCN_FEEDERREADY()	Determines if the feeder is ready to be advanced
SCN_FEEDERADVANCE()	Advances the feeder to the next document
SCN_SETUP()	Sets the scanning parameters
SCN_GETSIZE()	Returns scan line width (in bytes and pixels) and the number of scan lines
SCN_SCAN()	Tells the scanner driver to start scanning
SCN_ABORT()	Stops the scan and releases temporarily allocated resources
SCN_DIE()	Cleans up and calls exit()

SCN_INITOK() Function

```
arg->data: NULL
res->data: NULL
```

This function exists as a mechanism for the application to determine whether the scanner driver managed to initialize itself and the scanner properly. If any problem occurred during initialization, *res->errno* should be set to one of the SCE **#define**s in *scanner.h*; otherwise, no action is necessary.

SCN_PAGESIZE() Function

```
arg->data: int * (Metric)
res->data: SCWINDOW *

typedef struct tag_scwindow {
   float x, y, width, height;
} SCWINDOW;
```

This function returns the size of the scannable area supported by the scanner. Fill *res->data* in with the *x*, *y*, *width*, and *height* of the scannable area in inches or centimeters, depending on whether *arg->data* is SC_INCHES or SC_CENTIM.

SCN_MINMAXRES() Function

```
arg->data: NULL
res->data: SCMINMAXRES *

typedef struct tag_scminmaxres {
    float minx, miny, maxx, maxy;
} SCMINMAXRES;
```

This function sets the smallest and largest horizontal and vertical resolutions. *res->data->minx* should be set to the smallest horizontal resolution supported in hardware by the scanner, *res->data->miny* to the smallest vertical resolution, *res->data->maxx* to the largest horizontal resolution, and *res->data->maxy* to the largest vertical resolution.

SCN_NRES() Function

```
arg->data: NULL
res->data: int *
```

This function sets the number of resolution pairs supported in hardware. **res->data* should be set to the number of (*xres, yres*) resolution pairs supported in hardware by the scanner.

SCN_RES() Function

```
arg->data: int *
res->data: float *
```

This function sets floating-point numbers representing supported hardware resolutions. *arg->data* points to the metric of the resolution; either SC_INCHES for pixels/inch or SC_CENTIM for pixels/centimeter. *res->data* should be set to point to a floating-point array that represent supported hardware resolutions. There should be an even number of resolutions, with all of the horizontal resolutions first, then all of the vertical resolutions.

Note: All scanner drivers must support arbitrary resolutions; software routines are provided to perform zoom operations. The above information is provided so that scanner application developers can retrieve pure data from the scanner and perform their own zooming (with filters; *libscan* zooming does no filtering) to achieve the desired resolution.

SCN_NTYPES() Function

```
arg->data: NULL
res->data: int *
```

This function sets the number of data types supported by the driver. **res->data* should be set to the number of data types supported by this scanner driver.

SCN_TYPES() Function

```
arg->data: NULL
res->data: SCDATATYPE *

typedef struct tag_scdatatype {
    unsigned int packing : 4;
    unsigned int channels : 4;
    unsigned int type : 8;
    unsigned int bpp : 8;
} SCDATATYPE;
```

The *res->data* of the **SCN_TYPES()** function points to an array of **SCDATATYPE** objects, one for each of the types supported by the scanner driver.

All scanner drivers must support monochrome data; that is, the type
{ SC_PACKPIX, 1, SC_MONO, 1 }. All scanner drivers that support any kind of greyscale or color output must support the type { SC_PACKPIX, 1, SC_GREY, 8 }; that is, 8-bit gray-scale. All scanner drivers that support any kind of color output must support either { SC_PACKPIX, 3, SC_RGB, 8 } (24-bit CHUNKY color data) or { SC_PACKPLANE, 3, SC_RGB, 8 } (24-bit planar color data).

Library routines in *libscan.a* exist to facilitate compliance with these conventions.

SCN_FEEDERGETFLAGS() Function

```
arg->data: NULL
res->data: SCFEEDERFLAGS *

typedef unsigned int SCFEEDERFLAGS;
```

This function returns the feeder flags for this scanner to the application. See "Header Files" in Chapter 6.

SCN_FEEDERSETFLAGS() Function

```
arg->data: SCFEEDERFLAGS *
res->data: NULL
```

This function sets the feeder flags.

SCN_FEEDERREADY() Function

```
arg->data: NULL
res->data: NULL
```

This function determines whether or not the feeder is ready for an advance command.

SCN_FEEDERADVANCE() Function

```
arg->data: NULL
res->data: NULL
```

This function causes the feeder to advance to the next document.

SCN_SETUP() Function

```
arg->data: SCSETUP *
res->data: NULL

typedef struct tag_scsetup {
    int preview;
    SCDATATYPE type;
    int rmetric;
    float xres, yres;
    int wmetric;
    float x, y, width, height;
} SCSETUP;
```

The **SCN_SETUP()** function sets the scanning parameters. The upper-left *x* and *y* coordinates, the *width*, and the *height* are specified in either pixels, inches, or centimeters, depending on whether *arg->data->wmetric* is SC_PIXELS, SC_INCHES, or SC_CENTIM.

Set the scanning horizontal and vertical resolutions, in pixels per inch or pixels per centimeter, depending on the value of *arg->data->rmetric*. Set the data type for scanning. If this is a preview, *arg->data->preview* will have a nonzero value.

Note: If a resolution or combination of resolutions not supported in hardware is specified, the driver MUST zoom the image in order to supply the requested resolution. Library routines to aid zooming are available in *libscan.a*.

SCN_GETSIZE() Function

```
arg->data: NULL
res->data: SCSIZE *

typedef struct tag_scsize {
    long xbytes, xpixels, ysize;
} SCSIZE;
```

This function returns, to the scanning application, the width of a scan line in bytes and pixels, and the number of scan lines in the scan. This is called after **SCN_SETUP()** so the application knows exactly how much data to expect.

SCN_SCAN() Function

```
arg->data: NULL
res->data: NULL
```

This function tells the scanner driver to initiate scanning.

SCN_ABORT() Function

```
arg->data: NULL
res->data: NULL
```

This function stops the scan and releases any resources temporarily allocated. The application has decided to stop retrieving data before scanning has been completed. The driver should physically stop the scan and release any resources that were temporarily allocated for the scan.

SCN_DIE() Function

```
arg->data: NULL
res->data: NULL
```

This function cleans up and calls exit(2). The application has demanded that the driver terminate. This function should not return; it should perform any necessary cleanup and then call *exit*.

Type Conversion Macros

The macros listed in Table E-2 are provided to convert between data types.

Table E-2   Type Conversion Macros

Macro	Description
GRIDTOFLOAT	Convert from grid format to floating-point format
FLOATTOGRID	Convert from floating-point format to grid format

GRIDTOFLOAT and FLOATTOGRID Macros

GRIDTOFLOAT(int pos, int n)
FLOATTOGRID(float pos, int n)

These macros determine which destination pixel or line the source pixel or line at pos corresponds to. For example, if we are scanning at 120 dpi, but the application has requested 100 dpi, and our scan height is 1 inch, we need to skip 20 scan lines to provide the desired resolution. The following loop obtains scan lines from the scanner and passes them on to the application:

```
float fy;
int imgy, scany;

while (imgy < 100) {
    fy = GRIDTOFLOAT(imgy, 100);
    scany = FLOATTOGRID(fy, 120);
    ...
    /* Get the scany'th scan line from the scanner */
    /* Do conversion and horizontal zooming */
    /* Call SCDriverPutRow */
    imgy++;
}
```

Zooming and Type Conversion Functions

The functions listed in Table E-3 are provided to support zooming and converting between data types.

All conversion routines simultaneously zoom, so that only one conversion per line should ever be necessary.

Table E-3   Zooming and Type Conversion Functions

Function	Description
SCCreateZoomMap()	Creates a zoom map
SCDestroyZoomMap()	Frees memory allocated to store a zoom map
SCZoomRow1()	Zooms a row of 1-bit pixels
SCZoomRow8()	Zooms a row of 8-bit pixels
SCZoomRow24()	Zooms a row of 24-bit pixels
SCZoomRow32()	Zooms a row of 32-bit pixels
SCBandRGB8ToPixelRGB8()	Converts a row of pixels, in three rows (R, G, and B) of 8-bit components per pixel, to a row of 24-bit pixels
SCGrey8ToMono()	Converts a row of pixels from 8-bit greyscale to monochrome

SCCreateZoomMap() Function

```
int *SCCreateZoomMap(int anx, int bnx);
```

This function creates a zoom map. When zooming in the horizontal direction, it is wasteful to use **GRIDTOFLOAT** and **FLOATTOGRID** for every pixel of every line, since the same calculations would be repeated many times. A zoom map is an array of *bnx* integers, each of which is the pixel between 0 and *anx* - 1 that should be used when zooming a row of *anx* pixels to a row of *bnx* pixels. The zooming and conversion functions all take zoom maps for efficient zooming; for conversion functions where no zooming is to occur, the *zmap* parameter can be NULL.

SCDestroyZoomMap() Function

```
void SCDestroyZoomMap(int *zmap);
```

This function frees memory allocated to store a zoom map.

SCZoomRow1() Function

```
void SCZoomRow1(char *abuf, int anx, char *bbuf, int bnx, int *zmap);
```

This function zooms a row of *anx* pixels to a row of *bnx* pixels, 1 bit per pixel.

SCZoomRow8() Function

```
void SCZoomRow8(char *abuf, int anx, char *bbuf, int bnx, int *zmap);
```

This function zooms a row of *anx* pixels to a row of *bnx* pixels, 8 bits per pixel.

SCZoomRow24() Function

```
void SCZoomRow24(void *abuf, int anx, void *bbuf, int bnx, int *zmap);
```

This function zooms a row of *anx* pixels to a row of *bnx* pixels, 24 bits per pixel.

SCZoomRow32() Function

```
void SCZoomRow32(void *abuf, int anx, void *bbuf, int bnx, int *zmap);
```

This function zooms a row of *anx* pixels to a row of *bnx* pixels, 32 bits per pixel.

SCBandRGB8ToPixelRGB8() Function

```
void SCBandRGB8ToPixelRGB8(void *frombuf, int fromx,
        void *tobuf, int tox, int *zmap);
```

This function converts a row of *fromx* pixels, laid out in three rows (R, G, and B) of 8-bit components per pixel, to a row of *tox* pixels, 24 bits per pixel.

SCGrey8ToMono() Function

```
void SCGrey8ToMono(unsigned char thresh, void *frombuf,
        int fromx, void *tobuf, int tox, int *zmap);
```

This function converts a row of pixels from 8-bit greyscale to monochrome, thresholding each pixel with *thresh*.

Queues and Multi-Threaded Scanner Drivers

To achieve optimal performance in a scanner driver, it is helpful to parallelize the operations being performed. A pipeline carries data from the scanner to the ultimate destination, often a file. One can imagine that at the beginning of the pipeline, most of the time is spent waiting for I/O to complete. An intermediate image processing stage is CPU-bound as it zooms and converts rows of data. The final stage, writing to a file, is again I/O-bound.

Rather than adding these times together, we notice that all three stages of the pipeline can occur at the same time; that is, while the scanning stage is waiting for I/O, the file-writing stage can also be waiting for I/O, and the image-processing stage can be using the CPU. As you can imagine, performance gets even better on multiprocessor systems.

To support this, a multi-threaded queue interface is included in *libscan.a*. Each queue is semaphored so that the read thread can be different from the write thread, and so that the dequeue operation on an empty queue blocks until another thread has enqueued something.

In the driver template, separate threads implement the scanning stage and the image-processing stage. The main thread of the driver simply starts the two processes and waits for more commands from the application.

The driver template uses two queues: one to hold free buffers and one to hold freshly scanned lines. The amount of concurrency is metered by the initial size of the free queue; the scanning thread blocks when there are no more free buffers if it gets too far ahead of the image-processing thread.

The scanning thread dequeues a buffer from the scan free queue, gets data from the scanner, and stores it in the buffer. Then it breaks the buffer up into scan lines, enqueueing each line on the scan queue (it is typically faster to scan chunks of lines rather than one line at a time).

The image-processing thread dequeues a buffer from the scan queue. It then zooms and converts it, and writes the result to a stream that the application is reading to obtain the data. It puts the original buffer back on the scan free queue (actually, since the scanning thread breaks its buffer up into scan line-sized chunks, the image-processing thread has to know how to put the chunks back together).

Figure E-1 illustrates the scanning process.

Scanner
Application

Convert Process
writes cooked data
to the file descriptor
passed by the
application

Convert Process converts
from the scanner-specific
data formats to standard
formats and, if necessary,
performs simple zooming

Convert
Process

Scan
commands
and results

Convert Process adds
written buffers back
to the tail of
Free Queue

Convert Process gets
raw data buffers
from the head of
Scan Queue

Parent
Process

Free
Queue

Scan
Queue

Scan Process
gets the empty
buffers from the
head of Free Queue

Scan Process
adds the buffered
scanned data to the
tail of Scan Queue

Scan
Process

Scan Process issues
the device-specific
commands to get scan
data from the scanner

raw | data

Scanner

Figure E-1   Scanner Driver Architecture

Queue Manipulating Functions

The following functions are provided for manipulating queues:

```
typedef struct tag_scqueue SCQUEUE;
```

Table E-4 lists the queue manipulating functions.

Table E-4   Queue Manipulating Functions

Function	Description
SCCreateQueue()	Creates a queue that is multi-threaded safe and blocks on an empty dequeue
SCDestroyQueue()	Frees the resources used by a queue
SCEnqueue()	Adds an element to the tail of the queue
SCDequeue()	Removes an element from the head of a queue and returns it
SCQueueSetExit()	Sets a flag associated with a queue that tells all queue users to exit

SCCreateQueue() Function

```
SCQUEUE * SCCreateQueue(int nelems);
```

This function creates a queue that is multi-threaded safe and blocks on an empty dequeue. *nelems* is the maximum number of elements that can be stored in the newly created queue. Enqueue operations on full queues block until another thread has completed a dequeue operation.

SCDestroyQueue() Function

```
int SCDestroyQueue(SCQUEUE *q);
```

This function frees the resources used by a queue.

SCEnqueue() Function

```
void SCEnqueue(SCQUEUE *q, void *data);
```

This function adds an element to the tail of the queue. It unblocks a thread waiting to dequeue or blocks it if the queue is full.

SCDequeue() Function

```
void * SCDequeue(SCQUEUE *q);
```

This function removes an element from the head of a queue and returns it. **SCDequeue()** unblocks a thread waiting to enqueue or blocks it if the queue is empty.

SCQueueSetExit() Function

```
void * SCQueueSetExit(SCQUEUE *q);
```

This function sets a flag associated with a queue that tells all users of the queue to exit. Any thread blocking in **SCEnqueue()** or **SCDequeue()** is terminated. **SCQueueSetExit()** is used by the main thread of a scanner driver to tell the child threads to exit when the user aborts a scan.

Reference Pages

This appendix lists all reference pages associated with Impressario: the general interest reference pages, the printing developers reference pages, and the scanning developers reference pages.

Reference Pages

This appendix lists all reference pages associated with Impressario. Table F-1 lists the general interest reference pages.

Table F-1   General Interest Reference Pages

Names	Description
Impressario(1)	Printing and scanning environment for Silicon Graphics systems
glp(1), PrintPanel, printpanel	Graphical lp printing command
PrintStatus(1), printstatus	Graphical printer status tool
gscan(1)	Graphical scanning tool
scanners(1M)	Scanner installation and management tool
fileconvert(1)	File to printer or filetype converter
vstiff(1)	Stream TIFF viewer
psrip(1)	PostScript file to raster data format converter
installfoliofonts(1)	PostScript font installation program for Adobe Macintosh® Font Folio™ CDROM
installpcfonts(1)	PostScript font installation program for Adobe TypeSet™ PC floppy disks
printers(1M)	Printer installation and management program
print(1)	Printing subsystem, built on System V Release 3 printer spooling system, that uses Impressario

Table F-2 lists the printing developers reference pages and Table F-3 lists the scanning developers reference pages. The software these reference pages describe utilizes the C application program interface (API).

Table F-2   Printing Developers Reference Pages

Name	Description
libspool(3)	An API to the UNIX printer spooling systems.
libpod(3)	An API to the printer object database (POD).
libprintui(3X)	An API to the PrintBox widget.
libstiff(3)	An API for reading and writing the STIFF (Stream TIFF) data file format. It is described in detail in Appendix A.
libimp(3)	An API for reading and writing Silicon Graphics Image format files. It is described in detail in Appendix B.

Table F-3   Scanning Developers Reference Pages

Name	Description
libscan(3)	An API for scanning.
libstiff(3)	An API for reading and writing the STIFF (Stream TIFF) data file format. It is described in detail in Appendix A.
libimp(3)	An API for reading and writing Silicon Graphics image format files. It is described in detail in Appendix B.

Color Management in Impressario

This appendix explains the use of International Color Consortium (ICC) color profiles and PostScript Color Rendering Dictionaries (CRDs) in Impressario.

Color Management In Impressario

Impressario uses International Color Consortium (ICC) color profiles and PostScript Color Rendering Dictionaries (CRDs) for color management.

This appendix discusses the following topics:

- "An Overview" on page 254
- "ICC Color Profiles" on page 255
- "Color Rendering Dictionaries" on page 256
- "Generating CRDs and ICC Profiles" on page 257

An Overview

Color management is the process of specifying the color characteristics of peripheral devices and applying this information to allow color to be used more consistently throughout the system. Different devices have different color characteristics. The characteristics can be physical differences such as paper and ink for printers and phosphors for monitors. They also can be *color space* differences. A color space is a conceptual model for representing color.

Scanners and monitors usually use the RGB color space. Red, green, and blue are the primary colors in this space; all colors are mixtures of these. RGB is an additive color space. Red, green, and blue light are added to form a desired color. When full intensities of red, green, and blue light are used, the result is white light.

Printers use the CMY or CMYK color space. Cyan, magenta, and yellow are the primary colors. It is a subtractive color space: the color is that of the light that is reflected after parts of the white-light spectrum are absorbed by the medium, such as ink on paper. Theoretically, a mixture of cyan, magenta, and yellow subtractive colorings results in black because all of the white light is absorbed. In reality, a black color, denoted by *K*, is usually used because the CMY ink-on-paper mixture does not absorb light perfectly.

The technique of color management is to use color spaces to characterize input and output devices, and to provide the conversions needed to maintain consistent color use among them. Conversions may go directly from one color space to another, such as from RGB for a scanner to CMYK for a printer. Alternatively, conversions may make use of intermediary color spaces, such as CIE-based ones. (*CIE* stands for the Commission Internationale d'Eclairage.) The CIE color spaces are intended as representations of color as seen by the human eye. They are considered device-independent, in contrast to the RGB and CMYK spaces, whose color representations vary with the physical characteristics of the devices producing them. A common CIE-based space is CIELAB, which is based on the three types of stimuli to which the eye's retina responds. For a further discussion of color conversion, consult a text such as *Computer Graphics: Principles and Practice*, Second Edition, by Foley, van Dam, Feiner, and Hughes.

Impressario provides two tools for color management:

- ICC color profiles for SGI, TIFF, GIF, JPEG, PCD, and PPM raster image files
- PostScript CRDs for PostScript files that specify CIELAB-based color spaces

ICC Color Profiles

International Color Consortium(ICC) color profiles describe the color characteristics of source devices, such as scanners, and destination devices, such as printers. (A monitor is considered to be another type of device because it can be both a source and a destination.) Profiles are available for various scanners, printers, and monitors. By using the information from the source and destination profiles, a color correction can be determined and applied to the image.

The scanning program *gscan* adds the ICC source profile to its output (a TIFF or Silicon Graphics image raster file). Embedding the source profile in the output allows the file to be moved to systems that don't have that profile. With *gscan*, the user can choose from several scanner profiles, or even turn off color management altogether. The *Impressario User's Guide* has further information on scanning and *gscan*.

During printing, color correction is applied to a raster image by the *cocostiff* or *cocogif* utility before the image is processed by the PostScript interpreter. (*cocostiff* (Color Correct Stream Tiff) and *cocogif* (Color Correct GIF) are part of the *cms_eoe* subsystem) The Impressario file convert rules automatically invoke *cocostiff* or *cocogif*.

Use *fileconvert* to see how the file convert rules are applied:

```
/usr/sbin/fileconvert -d ImpressarioRasterBitmap
/usr/impressario/tests/data/testfile.sgi
```

returns

```
PRINTFILES="/usr/impressario/tests/data/testfile.sgi" ;
/usr/lib/print/sgi2stiff $IMPR_IMG2STIFFOPTS $PRINTFILES |
/usr/sbin/cocostiff $IMPR_CMGTOPTS  | /usr/lib/print/stiff2ps
$IMPR_SGI2PSOPTS | /usr/lib/print/psrip $IMPR_PSRIPOPTS
```

In this example, *sgi2stiff* converts the SGI format to the STIFF format, the output being piped to *cocostiff*. The command-line options of *cocostiff*, **-d hp650e06.pf** for example, are contained in the environment variable *$IMPR_CMGTOPTS*. This value, like those of the other uppercase environment variables, is normally set in a printer model file. (See the cocostiff(1) reference page for details about specific command-line options.) *cocostiff* applies the color correction, and its output is piped to *stiff2ps* and then to psrip, the PostScript interpreter.

It is important to note that an ICC profile describing the source device is needed to do color correction. As mentioned above, *gscan* embeds a source profile in the files it creates.

255

Of course, not all files are created by *gscan*. If a source profile is not embedded in the raster file, *cocostiff* and *cocogif* use the ICC color profile of a Sony monitor. The idea is for the default printed output to look similar to what would be displayed on the screen. Naturally, a source profile for a Sony monitor is needed to do this. *cocostiff* and *cocogif* also accept, as a command-line argument, a source profile that overrides the default profile and, if so specified, any embedded source profiles. See the cocostiff(1) and cocogif(1) reference pages for details.

ICC color profiles are stored in the */var/cms/profiles* directory. *cocostiff* and *cocogif* search this directory for any profile specified as a command-line argument and whose profile name does not include a full directory path.

For further information on ICC color profiles, see the web page at http://www.color.org. Search the web using the search string "International Color Consortium" to find several other technical references and applications.

Color Rendering Dictionaries

Color Rendering Dictionaries are used to manage the color of PostScript files, such as ones produced by Adobe Photoshop™, that specify CIELAB-based color spaces. PostScript files, such as those generated by ShowCase™, that do not specify a CIELAB-based color space do not have color management.

CRDs are stored in */usr/lib/print/data/CRDs*. A specific CRD is typically specified using the -**I** option with *psrip*. CRDs can be modified for use with the -**I** option by adding text at the end of the CRD to define it as a resource. An example of this is the last few lines from the */usr/lib/print/data/CRDs/hp1600C.crd* file:

```
/1600CCRD exch /ColorRendering defineresource pop
%%EndResource
{/1600CCRD /ColorRendering findresource } stopped
{ (%%[ Warning: Color CRD not found ]%%\r\n) print flush }
{dup /DefaultColorRendering exch /ColorRendering defineresource pop
setcolorrendering } ifelse
```

Tip: CRDs are usually binary files. If your editor can not edit the file, try using jot.

If a CRD is not specified with -**I,** a default is used. Default CRDs are in the directory */usr/lib/print/data/psrip/Resource/ColorRendering*. *ColorCRD* is the default color CRD and *MonoCRD* is the default for monochrome images.

Generating CRDs and ICC Profiles

When creating CRDs and ICC profiles, remember that many variables affect the color that is produced. These include the following:

- halftone screen or dithering (such as spot function or error diffusion)
- resolution (for example, 360 dpi or 720 dpi)
- paper type (plain, coated, transparencies, and so on)
- ink lot (changing inks can change output color)
- printer manufacturing variations
- age of the printers
- settings (draft, normal, presentation, microdot, and so on, depending on the printer)

Handling every case would require a separate CRD and ICC profile for each of the various combinations. In practice, this is rarely done because it requires a large number of CRDs and ICC profiles. A common approach, used by Impressario 2.0, is to generate one or two CRDs and profiles using the highest resolution, best paper, and best settings for the printer in question. The idea is to generate the CRDs and profiles for the printer when it is set to print in its best presentation mode.

Generating an ICC profile or a CRD typically consists of these steps:

1. Generate a test pattern containing a large number of color samples (say, 300 to 500).
2. Measure the color samples using a Colorimeter.
3. Generate several profiles and CRDs (for shadows, for highlights, and so on.)
4. Print sample images using the various CRDs and profiles, and select a preferred set.
5. Update the model file to use the preferred set.

This process requires specific software tools (and a lot of time). Impressario 2.0 does not include these tools, but third-party solutions are available. (If you use a third-party service or software package to generate ICC profiles or CRDs you should verify that you have the right to re-distribute them.)

Note: ColorSynergy® software by Candela generated all the CRDs and many of the ICC profiles for Impressario 2.*n*. A ColorTron™ colorimeter from Light Source was the input measuring device. For more information about these products contact:

Candela, Ltd.	Light Source Computer Images, Inc.
1676 East Cliff Road	17 E. Sir Francis Drake Suite 100
Burnsville, MN 55337-1300	Larkspur, CA 94939
612 894-8890	415 925-4200
612 894-8840 (fax)	415 461-8011 (fax)
	URL: http://www.ls.com

As a final step, look at the way ICC profiles and CRDs are used in the model file */var/spool/lp/model/deskjetII_model*. At the beginning of this file, default ICC profiles and CRDs are set for every printer. When the script is run, it calls the function **get_icc_crd_name()** to determine names for the ICC profile and CRD based on the printer model, paper type, and halftone being used. For example, if the printer is an HP DeskJet 1600C, **get_icc_crd_name()** returns two of the following names (one for an ICC profile, the other for a CRD):

ICC Profile Name	CRD Name
HP_DeskJet_1600C_special_spot.pf	HP_DeskJet_1600C_special_spot.crd
HP_DeskJet_1600C_special_diffused.pf	HP_DeskJet_1600C_special_diffused.crd
HP_DeskJet_1600C_glossy_spot.pf	HP_DeskJet_1600C_glossy_spot.crd
HP_DeskJet_1600C_glossy_diffused.pf	HP_DeskJet_1600C_glossy_diffused.crd

It is the command returned by *fileconvert*, later in the model file, that determines whether the ICC profile or CRD is used. If the file to be printed is a raster image, color correction is performed using an ICC profile. If the HP 1600C is using glossy paper and error diffusion, and if an ICC profile in the */var/cms/profiles* directory has the name *HP_DeskJet_1600C_glossy_diffused.pf*, the model file will find and use it. Otherwise it uses a default. After color correction, the file to be printed is converted to a PostScript file that is not CIELAB-based, and the PostScript file is sent to the PostScript interpreter.

If the file to be printed is already PostScript, it is sent directly to the PostScript interpreter where a CRD is then applied if the file is CIELAB-based. As before, a default CRD is used if one with the proper name is not found. Notice that a CRD is never used on a PostScript file that has undergone color correction with an ICC profile.

If you implement a function similar to **get_icc_crd_name()**, you should document the ICC profiles and CRD names that the driver recognizes. This allows users who create their own ICC profiles and CRDs to install them in the correct directories using the correct names. The goal is to enable users to supply ICC profiles and CRDs specific to the printers, inks, paper stocks, and so on in their environment.

Using the Adobe Configurable PostScript Interpreter

This appendix explains legal restrictions concerning the use of the Adobe Configurable PostScript Interpreter.

Using the Adobe Configurable PostScript Interpreter

Please note that Impressario and the Impressario Developer's Kit (collectively, "IMPR") includes Adobe Systems, Inc. Configurable PostScript Interpreter ("CPSI"), also known as "psrip," and is subject to the restrictions set forth below, as well as those set forth in the Silicon Graphics, Inc. Software License Agreement that accompanies Impressario.

1. IMPR may be used to develop device drivers for any of the output devices specified in Table H-1.

Table H-1   Permitted IMPR Device Drivers

Company/Manufactured	Designated Output Device
Canon®	CJ-10
Epson	Color Stylus, Stylus Pro, Stylus Pro XL, Color Stylus II, Color Stylus IIs, 9 pin, 24 pin
Fargo	Primera®
Hewlett-Packard	LaserJet IIP, IIP+, III, IIIP, 4, 4L, 4P, 4 Plus, 4V, 4Si, 5L, 5P, 5Si DeskJet 500C, 550C, 560C, 1200C, 660C, 850C, 855C, 1600C DesignJet 250C, 650C, 750C PaintJet XL300 CopyJet™
Genicom	9160, 9170-1, 9170-2, 9080
Seiko	ColorPoint™ CH6104, ColorPoint 2™ CH7214, ColorPoint 2 CH7204-RSF
Tektronix®	Phaser® II SX, Phaser III RX, Phaser 300 RX, Phaser II SD, Phaser II SDX

2. IMPR may be used to develop device drivers for any output device not listed in item 1 above, provided that such output device has a retail price of less than three thousand five hundred dollars ($3,500) U.S., *and* has an image data content of less then or equal to 600 bytes per inch, where image data content is determined by the formula below:

```
Image data Content (Bytes/Inch) = (Mres x B x C) / 8
```

where

```
Mres = Maximum device resolution in pixels per inch
B    = number of bits per colorant (the -B option to psrip)
C    = number of colorants per pixel (RGB would be 3, CMYK 4, etc.)
```

Note: IF YOUR DRIVER DOES NOT MEET THE REQUIREMENTS SET FORTH ABOVE THEN YOU DO NOT HAVE THE RIGHT TO USE OR DISTRIBUTE SUCH DRIVER IN CONJUNCTION WITH CPSI AS CONTAINED IN IMPR. IF YOU WISH TO REQUEST ADDITIONAL RIGHTS PLEASE CONTACT:

Silicon Graphics, Inc.
Digital Media Systems Division
Legal Department
2011 North Shoreline Blvd.
Mountain View, CA 94043
415 960-1980
URL: http://www.sgi.com

Glossary

Glossary

API

Application program interface; a set of function calls for achieving some purpose.

BSD

Berkeley Software Distribution.

Chunky data format

The original Impressario 1.0 data format; it has been made obsolete in favor of the Stream TIFF data format.

CMY STIFF data format

CMY data class is a subset of the CMYK class and differs from the CMYK class in a TIFF-compliant manner. See "Generic STIFF File Structure" in Appendix A for detailed information.

CMYK STIFF data format

CMYK is a TIFF data format extension. See "Generic STIFF File Structure" in Appendix A for detailed information. CMYK stands for cyan, magenta, yellow, and black, the subtractive color process primaries.

CPSI

Configurable PostScript Interpreter. Licensed from Adobe Systems, Inc., it is the foundation of psrip.

File Type Rules (FTR)

A database that is one of the three key components of the Impressario file conversion pipeline. These rules are well-documented in the *Indigo Magic Desktop Integration Guide*, available as an online book and installable from your IRIX CD.

filter/driver specification

See "The Filter/Driver Specification and psrip" in Chapter 2.

FTR

See File Type Rules.

generic scanner interface

An interface between application programs and scanner drivers. See Chapter **8**, "Generic Scanner Interface," for additional information.

GIF

Graphics Interchange Format. A format for storing multibit images and graphics.

glp

A graphical end-user interface for submitting print jobs from applications. See the glp(1) reference page for additional information.

gscan

A graphical end-user interface for using scanners. See the gscan(1) reference page for additional information.

GUI

Graphical user interface.

Impressario

A visual printing and scanning environment for IRIS workstations.

JPEG

Joint Photographic Experts Group. A file format that contains a JPEG data stream.

libimp

A C application program interface (API) for reading and writing Silicon Graphics Image Format files.

libpod

A C application program interface to the printer object database (POD).

libprintui

A graphical user interface (GUI) printing library.

libscan

A C application program interface for scanning.

libspool

A C application program interface to the UNIX printer spooling system.

libstiff

A C-language API for reading and writing Stream TIFF files.

lp command

The System V release 3 command to send a print job to the printer. See the lp(1) reference page.

lpr command

The BSD command to send a print job to a printer. See the lpr(1) reference page.

lpsched command

The System VR3 spooler job scheduler. Not invoked by end users.

packed data format

Bitmapped data organized by pixel, with all color components for each pixel adjacent. For example: RGBRGBRGB. . .

planar data format

Bitmapped data organized by color "plane," with all pixels arranged in planes of component colors, all components of one color, then another, and so on.

PCD

Kodak Photo CD. Files using Eastman Kodak's proprietary format for storing digital images.

plotter format

A raster-based page-description language, most commonly HPGL or some variant.

POD

See printer object database.

PostScript printer

A printer with a built-in PostScript interpreter. It prints only PostScript files.

PPM

Portable Pixmap Utilities. A format used in color bitmap file conversion.

print client

Any system other than the print server that wishes to use a printer.

print server

The system that controls a printer.

PrintBox

A graphical end-user interface for submitting print jobs from applications. See the libprintui(3) reference page for additional information.

Printer Manager

A graphical end-user interface for managing and installing printers. See the reference page for additional information.

printer model file

A Bourne shell script that controls the filtering and printing of a set of files. Invoked by the *lpsched* command. See Chapter 3, "Printer Model Files."

<printer name>.config

A configuration file representing the capabilities of the printer. The *.config* file is part of the printer object database (POD).

<printer name>.log

The printer log file for the specified printer. The *.log* file is part of the printer object database (POD).

<printer name>.status

A status file indicating the current printing state. The *.status* file is part of the printer object database (POD).

printer object database (POD)

The POD contains information on the current configuration, status, and job history of a single printer. It is the central database used by all printing filters and is maintained by each driver. See Appendix C, "Printer Object Database (POD) File Formats," for detailed information.

printers

See Printer Manager.

PrintPanel

An alias for *glp*, a graphical end-user interface for modifying printer settings. See the glp(1) reference page for additional information.

PrintStatus

A graphical end-user interface for checking printer status. See the PrintStatus(1) reference page for additional information.

raster printer

A printer that accepts only bitmap image data.

scanner driver

A program that interfaces with a scanner. See Chapter 6, "Scanner Drivers," for additional information.

scanners

A graphical end-user interface for installing and managing scanners. See the scanners(1) reference page for additional information.

SGI

Silicon Graphics, Inc. image file format. Format for storing black-and-write, color RGB, and color RGB with alpha channel images.

spooling system API

See libspool and libprintui.

STIFF

Stream TIFF is a subset of the Tagged Image File Format (TIFF) originally developed by Aldus Corporation. See Appendix A, "Stream TIFF Data Format," for more information.

System V spooler interface

See lp command and lpsched command.

Tagged Image File Format

The Tagged Image File Format (TIFF) originally developed by Aldus Corporation. See Appendix A, "Stream TIFF Data Format," for more information.

TIFF

See Tagged Image File Format.

YMC data format

A data class similar to the CMY class with the exception that data is organized as YMC instead of CMY. YMC stands for yellow, magenta, and cyan.

YMCK data format

A class similar to CMYK except that data is organized as YMCK instead of CMYK. YMCK stands for yellow, magenta, cyan, and black.

Index

Index