

Devices_Manual

COLLABORATORS

	<i>TITLE :</i> Devices_Manual		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		July 18, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Devices_Manual	1
1.1	Amiga® RKM Devices: 10 Printer Device	1
1.2	10 Printer Device / Printer Device Commands and Functions	1
1.3	10 Printer Device / Printer Device Access	2
1.4	10 / Printer Device Access / Opening Prt:	3
1.5	10 / Printer Device Access / Writing To Prt:	4
1.6	10 / Printer Device Access / Closing Prt:	4
1.7	10 Printer Device / Device Interface	5
1.8	10 / Device Interface / Opening The Printer Device	6
1.9	10 / Device Interface / Writing Text To The Printer Device	7
1.10	10 / Device Interface / Important Points About Print Requests	8
1.11	10 / Device Interface / Closing The Printer Device	8
1.12	10 Printer Device / Sending Printer Commands to a Printer	9
1.13	10 / Sending Printer Commands to a Printer / Command Definitions	9
1.14	10 Printer Device / Obtaining Printer Specific Data	12
1.15	10 Printer Device / Reading and Changing Printer Preferences Settings	12
1.16	10 Printer Device / Querying the Printer Device	13
1.17	10 Printer Device / Error Codes from the Printer Device	14
1.18	10 Printer Device / Dumping a Rastport to a Printer	15
1.19	10 / Dumping a Rastport to a Printer / Printer Special Flags	16
1.20	10 / Dumping Rastport to a Printer / Printing Corrected Aspect Ratio	17
1.21	10 / Dumping a Rastport to a Printer / Strip Printing	18
1.22	10 / Dumping Rastport to a Printer / Additional Notes on Graphic Dumps	19
1.23	10 Printer Device / Creating a Printer Driver	19
1.24	10 / Creating a Printer Driver / Writing Alphanumeric Printer Drivers	21
1.25	10 // Writing An Alphanumeric Printer Driver / Command Table	22
1.26	10 // Writing An Alphanumeric Printer Driver / DoSpecial()	22
1.27	10 // Writing An Alphanumeric Printer Driver / Printertag.asm	24
1.28	10 // Writing An Alphanumeric Printer Driver / Extended Character Table	24
1.29	10 // Writing An Alphanumeric Printer Driver / Character Conversion	25

1.30	10 / Creating a Printer Driver / Writing A Graphics Printer Driver	27
1.31	10 / Writing A Graphics Printer Driver / Render()	27
1.32	10 / Writing A Graphics Printer Driver / Transfer()	31
1.33	10 / Writing A Graphics Printer Driver / SetDensity()	33
1.34	10 / Writing A Graphics Printer Driver / Printertag.asm	33
1.35	10 / Creating a Printer Driver / Testing The Printer Driver	34
1.36	10 Printer Device / Example Printer Driver Source Code	35
1.37	10 / Example Printer Driver Source Code / EpsonX	35
1.38	10 / Example Printer Driver Source Code / HP_Laserjet	36
1.39	10 Printer Device / Additional Information on the Printer Device	36

Chapter 1

Devices_Manual

1.1 Amiga® RKM Devices: 10 Printer Device

The printer device offers a way of sending configuration-independent output to a printer attached to the Amiga. It can be thought of as a filter: it takes standard commands as input and translates them into commands understood by the printer. The commands sent to the printer are defined in a specific printer driver program. For each type of printer in use, a driver (or the driver of a compatible printer) should be present in the `devs:printers` directory.

Printer Driver Source Code In This Chapter

```
-----
EpsonX           A YMCB, 8 pin, multi-density interleaved printer.
HP_LaserJet       A black and white, multi-density, page-oriented printer.
```

```
Printer Device Commands and Functions
Printer Device Access
Device Interface
Sending Printer Commands to a Printer
Obtaining Printer Specific Data
Reading and Changing the Printer Preferences Settings
Querying the Printer Device
Error Codes from the Printer Device
Dumping a Rastport to a Printer
Creating a Printer Driver
Example Printer Driver Source Code
Additional Information on the Printer Device
```

1.2 10 Printer Device / Printer Device Commands and Functions

Command	Operation
-----	-----
CMD_FLUSH	Remove all queued requests for the printer device. Does not affect active requests.
CMD_RESET	Reset the printer device to its initialized state. All active and queued I/O requests will be aborted.

CMD_START	Restart all paused I/O requests
CMD_STOP	Pause all active and queued I/O requests.
CMD_WRITE	Write out a stream of characters to the printer device. The number of characters can be specified or a NULL-terminated string can be sent.
PRD_DUMPRPORT	Dump the specified RastPort to a graphics printer.
PRD_PRTCOMMAND	Send a command to the printer.
PRD_QUERY	Return the status of the printer port's lines and registers.
PRD_RAWWRITE	Send unprocessed output to the the printer.

Exec Functions as Used in This Chapter

AbortIO()	Abort a command to the printer device.
CloseDevice()	Relinquish use of the printer device. All requests must be complete before closing.
DoIO()	Start a command and wait for completion (synchronous request).
OpenDevice()	Obtain use of the printer device.
SendIO()	Start a command and return immediately (asynchronous request).
WaitIO()	Wait for the completion of an asynchronous request. When the request is complete, the message will be removed from the printer message port.

Exec Support Functions as Used in This Chapter

CreatePort()	Create a signal message port for reply messages from the audio device. Exec will signal a task when a message arrives at the reply port.
CreateExtIO()	Create an I/O request structure of type printerIO. This structure will be used to send commands to the printer device.
DeletePort()	Delete the message port created by CreatePort().
DeleteExtIO()	Delete an I/O request structure created by CreateExtIO().

1.3 10 Printer Device / Printer Device Access

The printer device is totally transparent to an application. It uses information set up by the Workbench Preferences Printer and PrinterGfx tools to identify the type of printer connection (serial or parallel), type of dithering, etc. It also offers the flexibility to send raw information to the printer for special non-standard or unsupported features. Raw data transfer is not recommended for conventional text and graphics since it will result in applications that will only work with certain printers. By using the standard printer device interface, an application can perform device independent output to a printer.

Don't Hog The Device.

The printer device is currently an exclusive access device. Do not tie it up needlessly.

There are two ways of doing output to the printer device:

- * PRT:-the AmigaDOS printer device
PRT: may be opened just like any other AmigaDOS file. You may send standard escape sequences to PRT: to specify the options you want as shown in the command table below. The escape sequences are interpreted by the printer driver, translated into printer-specific escape sequences and forwarded to the printer. When using PRT: the escape sequences and data must be sent as a character stream. Using PRT: is by far the easiest way of doing text output to a printer.
- * printer.device - to directly access the printer device itself
By opening the printer device directly, you have full control over the printer. You can either send standard escape sequences as shown in the command table below or send raw characters directly to the printer with no processing at all. Doing this would be similar to sending raw characters to SER: or PAR: from AmigaDOS. (Since this interferes with device-independence it is strongly discouraged). Direct access to the printer device also allows you to transmit device I/O commands, such as reset and flush, and do a raster dump on a graphics-capable printer.

Use A Stream to Escape.

All "raw escape sequences" transmitted to the printer through the printer device must take the form of a character stream.

Opening Prt: Writing To Prt: Closing Prt:

1.4 10 / Printer Device Access / Opening Prt:

When using the printer device as PRT:, you can open it just as though it were a normal AmigaDOS output file.

```
struct FileHandle *file;

file = Open( "PRT:", MODE_NEWFILE ); /* Open PRT: */
if (file == 0)                       /* if the open was unsuccessful */
    exit(PRINTER_WONT_OPEN);
```

1.5 10 / Printer Device Access / Writing To Prt:

Once you've opened it, you can print by calling the AmigaDOS Write() standard I/O routine.

```
actual_length = Write(file, dataLocation, length);
```

where

file is a file handle.

dataLocation

is a pointer to the first character in the output stream you wish to write. This stream can contain the standard escape sequences as shown in the command table below. The printer command aRAW (see the Printer Device Command Functions table below) can be used in the stream if character translation is not desired.

length

is the length of the output stream.

actual_length

is the actual length of the write. For the printer device, if there are no errors, this will be the same as the length of write requested. The only exception is if you specify a value of -1 for length. In this case, -1 for length means that a null (0) terminated stream is being written to the printer device. The device returns the count of characters written prior to encountering the null. If it returns a value of -1 in actual_length, there has been an error.

-1 = STOP!

If a -1 is returned by Write(), do not do any additional printing.

1.6 10 / Printer Device Access / Closing Prt:

When the printer I/O is complete, you should close PRT:. Don't keep the device open when you are not using it. The user may have changed the printer settings by using the Workbench Preferences tool. There's also the possibility the printer has been turned off and on again causing the printer to switch to its own default settings. Every time the printer device is opened, it reads the current Preferences settings. Hence, by always opening the printer device just before printing and always closing it afterwards, you ensure that your application is using the current Preferences settings.

```
Close(file);
```

In DOS, You Must Be A Process.

Printer I/O through the DOS must be done by a process, not by a task. DOS utilizes information in the process control block and would become confused if a simple task attempted to perform these activities. Printer I/O using the printer device directly, however,

can be performed by a task.

The remainder of this chapter will deal with using the printer device directly.

1.7 10 Printer Device / Device Interface

The printer device operates like the other Amiga devices. To use it, you must first open the printer device, then send I/O requests to it, and then close it when finished. See the Introduction to Amiga Devices chapter for general information on device usage.

There are three distinct kinds of data structures required by the printer I/O routines. Some of the printer device I/O commands, such as CMD_START and CMD_WRITE require only an IOStdReq data structure. Others, such as PRD_DUMPREPORT and PRD_PRTCOMMAND, require an extended data structure called IODRPreq (for "Dump a RastPort Request") or IOPrtCmdReq (for "Printer Command Request").

For convenience, it is strongly recommended that you define a single data structure called printerIO, that can be used to represent any of the three pre-defined printer communications request blocks.

```
union printerIO
{
    struct IOStdReq    ios;
    struct IODRPreq    iodrp;
    struct IOPrtCmdReq iopc;
};

struct IODRPreq
{
    struct Message io_Message;
    struct Device *io_Device;      /* device node pointer */
    struct Unit   *io_Unit;        /* unit (driver private) */
    UWORD io_Command;              /* device command */
    UBYTE io_Flags;
    BYTE io_Error;                 /* error or warning num */
    struct RastPort *io_RastPort;  /* raster port */
    struct ColorMap *io_ColorMap;  /* color map */
    ULONG io_Modes;                /* graphics viewport modes */
    UWORD io_SrcX;                 /* source x origin */
    UWORD io_SrcY;                 /* source y origin */
    UWORD io_SrcWidth;             /* source x width */
    UWORD io_SrcHeight;            /* source x height */
    LONG io_DestCols;              /* destination x width */
    LONG io_DestRows;              /* destination y height */
    UWORD io_Special;              /* option flags */
};

struct IOPrtCmdReq
{
    struct Message io_Message;
    struct Device *io_Device;      /* device node pointer */
    struct Unit   *io_Unit;        /* unit (driver private) */
};
```

```

        UWORD   io_Command;           /* device command */
        UBYTE   io_Flags;
        BYTE    io_Error;             /* error or warning num */
        UWORD   io_PrtCommand;        /* printer command */
        UBYTE   io_Parm0;             /* first command parameter */
        UBYTE   io_Parm1;             /* second command parameter */
        UBYTE   io_Parm2;             /* third command parameter */
        UBYTE   io_Parm3;             /* fourth command parameter */
    };

```

See the include file `exec/io.h` for more information on `IOStdReq` and the include file `devices/printer.h` for more information on `IODRPReq` and `IOPrtCmdReq`.

Opening The Printer Device
 Writing Text To The Printer Device
 Important Points About Print Requests
 Closing The Printer Device

1.8 10 / Device Interface / Opening The Printer Device

Three primary steps are required to open the printer device:

- * Create a message port using `CreatePort()`. Reply messages from the device must be directed to a message port.
- * Create an extended I/O request structure of type `printerIO` with the `CreateExtIO()` function. This means that one memory area can be used to represent three distinct forms of memory layout for the three different types of data structures that must be used to pass commands to the printer device. By using `CreateExtIO()`, you automatically allocate enough memory to hold the largest structure in the union statement.
- * Open the printer device. Call `OpenDevice()`, passing the I/O request.

```

union printerIO
{
    struct IOStdReq   ios;
    struct IODRPReq   iodrp;
    struct IOPrtCmdReq iopc;
};

struct MsgPort *PrintMP;           /* Message port pointer */
union printerIO *PrintIO;          /* I/O request pointer */

if (PrintMP=CreateMsgPort() )
    if (PrintIO=(union printerIO *)
        CreateExtIO(PrintMP,sizeof(union printerIO)) )
        if (OpenDevice("printer.device",0L,(struct IORequest *)PrintIO,0))
            printf("printer.device did not open\n");

```

The printer device automatically fills in default settings for all printer device parameters from Preferences. In addition, information about the printer itself is placed into the appropriate fields of `printerIO`. (See

the Obtaining Printer Specific Data section below.)

Pre-V36 Tasks and OpenDevice().

Tasks in pre-V36 versions of the operating system are not able to safely OpenDevice() the printer device because it may be necessary to load it in from disk, something only a process could do under pre-V36. V36 and higher versions of the operating system do not have such a limitation.

1.9 10 / Device Interface / Writing Text To The Printer Device

Text written to a printer can be either processed text or unprocessed text.

Processed text is written to the device using the CMD_WRITE command. The printer device accepts a character stream, translates any embedded escape sequences into the proper sequences for the printer being used and then sends it to the printer. The escape sequence translation is based on the printer driver selected either through Preferences or through your application. You may also send a NULL-terminated string as processed text.

Unprocessed text is written to the device using the PRD_RAWWRITE command. The printer device accepts a character stream and sends it unchanged to the printer. This implies that you know the exact escape sequences required by the printer you are using. You may not send a NULL-terminated string as unprocessed text.

One additional point to keep in mind when using PRD_RAWWRITE is that Preference settings for the printer are ignored. Unless the printer has already been initialized by another command, the printer's own default settings will be used when printing raw, not the user's Preferences settings.

You write processed text to the printer device by passing an IOStdReq to the device with CMD_WRITE set in io_Command, the number of bytes to be written set in io_Length and the address of the write buffer set in io_Data.

To write a NULL-terminated string, set the length to -1; the device will output from your buffer until it encounters a value of zero (0x00).

```
PrintIO->ios.io_Length = -1;
PrintIO->ios.io_Data    =
    (APTR)"I went to a fight and a hockey game broke out."
PrintIO->ios.io_Command = CMD_WRITE;
DoIO((struct IORequest *)PrintIO);
```

The length of the request is -1, meaning we are writing a NULL-terminated string. The number of characters sent will be found in io_Actual after the write request has completed.

You write unprocessed text to the printer device by passing an IOStdReq to the device with PRD_RAWWRITE set in io_Command, the number of bytes to be written set in io_Length and the address of the write buffer set in io_Data.

```
UBYTE *outbuffer;
```

```
PrintIO->ios.io_Length   = strlen(outbuffer);
PrintIO->ios.io_Data     = (APTR)outbuffer;
PrintIO->ios.io_Command  = PRD_RAWWRITE;
DoIO((struct IORequest *)PrintIO);
```

IOStdReq Only.

I/O requests with CMD_WRITE and PRD_RAWWRITE must use the IOStdReq structure of the union printerIO.

1.10 10 / Device Interface / Important Points About Print Requests

- * Perform printer I/O from a separate task or process
It is quite reasonable for a user to expect that printing will be performed as a background operation. You should try to accommodate this expectation as much as possible.
- * Give the user a chance to stop
Your application should always allow the user to stop a print request before it is finished.
- * Don't confuse aborting a print request with cancelling a page
Some applications seem to offer the user the ability to abort a multi-page print request when in fact the abort is only for the current page being printed. This results in the next page being printed instead of the request being stopped. Do not do this! It only confuses the user and takes away from your application. There is nothing wrong with allowing the user to cancel a page and continue to the next page, but it should be explicit that this is the case. If you abort a print request, the entire request should be aborted.

1.11 10 / Device Interface / Closing The Printer Device

Each OpenDevice() must eventually be matched by a call to CloseDevice().

All I/O requests must be complete before CloseDevice(). If any requests are still pending, abort them with AbortIO().

```
AbortIO(PrintIO); /* Ask device to abort request, if pending */
WaitIO(PrintIO); /* Wait for abort, then clean up */
```

```
CloseDevice((struct IORequest *)PrintIO);
```

Use AbortIO() / WaitIO() Intelligently.

Only call AbortIO()/WaitIO() for requests which have already been sent to the printer device. Using the AbortIO()/WaitIO() sequence on requests which have not been sent results in a hung condition.

1.12 10 Printer Device / Sending Printer Commands to a Printer

As mentioned before, it is possible to include printer commands (escape sequences) in the character stream and send them to the printer using the CMD_WRITE device I/O command. It is also possible to use the printer command names using the device I/O command PRD_PRTCOMMAND with the IOPrtCmdReq data structure. This gives you a mnemonic way of setting the printer to your program needs.

You send printer commands to the device by passing an IOPrtCmdReq to the device with PRD_PRTCOMMAND set in io_Command, the printer command set in io_PrtCommand and up to four parameters set in Parm0 through Parm3.

```
#include <devices/printer.h>

PrintIO->iopc.io_PrtCommand = aSLRM; /* Set left & right margins */
PrintIO->iopc.io_Parm0 = 1;          /* Set left margin = 1 */
PrintIO->iopc.io_Parm1 = 79;         /* Set right margin = 79 */
PrintIO->iopc.io_Parm2 = 0;
PrintIO->iopc.io_Parm3 = 0;
PrintIO->iopc.io_Command = PRD_PRTCOMMAND;
DoIO((struct IOREquest *)PrintIO);
```

Consult the command function table listed below for other printer commands.

Printer Command Definitions

1.13 10 / Sending Printer Commands to a Printer / Command Definitions

The following table describes the supported printer functions.

Just Because We Have It Doesn't Mean You Do.

Not all printers support every command. Unsupported commands will either be ignored or simulated using available functions.

To transmit a command to the printer device, you can either formulate a character stream containing the material shown in the "Escape Sequence" column of the table below or send an PRD_PRTCOMMAND device I/O command to the printer device with the "Name" of the function you wish to perform.

PRINTER DEVICE COMMAND FUNCTIONS

Name	Cmd No.	Escape Sequence	Function	Defined by:
----	----	-----	-----	-----
aRIS	0	ESCc	Reset	ISO
aRIN	1	ESC#1	Initialize	+++
aIND	2	ESCD	Linefeed	ISO
aNEL	3	ESCE	Return, linefeed	ISO
aRI	4	ESCM	Reverse linefeed	ISO
aSGR0	5	ESC[0m	Normal char set	ISO
aSGR3	6	ESC[3m	Italics on	ISO

aSGR23	7	ESC[23m	Italics off	ISO
aSGR4	8	ESC[4m	Underline on	ISO
aSGR24	9	ESC[24m	Underline off	ISO
aSGR1	10	ESC[1m	Boldface on	ISO
aSGR22	11	ESC[22m	Boldface off	ISO
aSFC	12	ESC[nm	Set foreground color where n stands for a pair of ASCII digits, 3 followed by any number 0-9 (See ISOCOLOR Table)	ISO
aSBC	13	ESC[nm	Set background color where n stands for a pair of ASCII digits, 4 followed by any number 0-9 (See ISO Color Table)	ISO
aSHORP0	14	ESC[0w	Normal pitch	DEC
aSHORP2	15	ESC[2w	Elite on	DEC
aSHORP1	16	ESC[1w	Elite off	DEC
aSHORP4	17	ESC[4w	Condensed fine on	DEC
aSHORP3	18	ESC[3w	Condensed off	DEC
aSHORP6	19	ESC[6w	Enlarged on	DEC
aSHORP5	20	ESC[5w	Enlarged off	DEC
aDEN6	21	ESC[6"z	Shadow print on	DEC
aDEN5	22	ESC[5"z	Shadow print off (sort of)	DEC
aDEN4	23	ESC[4"z	Doublestrike on	DEC
aDEN3	24	ESC[3"z	Doublestrike off	DEC
aDEN2	25	ESC[2"z	NLQ on	DEC
aDEN1	26	ESC[1"z	NLQ off	DEC
aSUS2	27	ESC[2v	Superscript on	+++
aSUS1	28	ESC[1v	Superscript off	+++
aSUS4	29	ESC[4v	Subscript on	+++
aSUS3	30	ESC[3v	Subscript off	+++
aSUS0	31	ESC[0v	Normalize the line	+++
aPLU	32	ESCL	Partial line up	ISO
aPLD	33	ESCK	Partial line down	ISO
aFNT0	34	ESC(B	US char set or Typeface 0	DEC
aFNT1	35	ESC(R	French char set or Typeface 1	DEC
aFNT2	36	ESC(K	German char set or Typeface 2	DEC
aFNT3	37	ESC(A	UK char set or Typeface 3	DEC
aFNT4	38	ESC(E	Danish I char set or Typeface 4	DEC
aFNT5	39	ESC(H	Swedish char set or Typeface 5	DEC
aFNT6	40	ESC(Y	Italian char set or Typeface 6	DEC
aFNT7	41	ESC(Z	Spanish char set or Typeface 7	DEC
aFNT8	42	ESC(J	Japanese char set or Typeface 8	+++
aFNT9	43	ESC(6	Norwegian char set or Typeface 9	DEC
aFNT10	44	ESC(C	Danish II char set or Typeface 10 (See Suggested Typefaces Table)	+++
aPROP2	45	ESC[2p	Proportional on	+++
aPROP1	46	ESC[1p	Proportional off	+++
aPROP0	47	ESC[0p	Proportional clear	+++
aTSS	48	ESC[n E	Set proportional offset	ISO
aJFY5	49	ESC[5 F	Auto left justify	ISO
aJFY7	50	ESC[7 F	Auto right justify	ISO
aJFY6	51	ESC[6 F	Auto full justify	ISO

aJFY0	52	ESC[0 F	Auto justify off	ISO
aJFY3	53	ESC[3 F	Letter space (justify)	(special)ISO
aJFY1	54	ESC[1 F	Word fill(auto center)	(special)ISO
aVERP0	55	ESC[0z	1/8" line spacing	+++
aVERP1	56	ESC[1z	1/6" line spacing	+++
aSLPP	57	ESC[nt	Set form length n	DEC
aPERF	58	ESC[nq	Perf skip n (n>0)	+++
aPERF0	59	ESC[0q	Perf skip off	+++
aLMS	60	ESC#9	Left margin set	+++
aRMS	61	ESC#0	Right margin set	+++
aTMS	62	ESC#8	Top margin set	+++
aBMS	63	ESC#2	Bottom margin set	+++
aSTBM	64	ESC[n;	nr Top and bottom margins	DEC
aSLRM	65	ESC[n;	ns Left and right margins	DEC
aCAM	66	ESC#3	Clear margins	+++
aHTS	67	ESCH	Set horizontal tab	ISO
aVTS	68	ESCJ	Set vertical tabs	ISO
aTBC0	69	ESC[0g	Clear horizontal tab	ISO
aTBC3	70	ESC[3g	Clear all h. tabs	ISO
aTBC1	71	ESC[1g	Clear vertical tab	ISO
aTBC4	72	ESC[4g	Clear all v. tabs	ISO
aTBCALL	73	ESC#4	Clear all h. & v. tabs	+++
aTBSALL	74	ESC#5	Set default tabs	+++
aEXTEND	75	ESC[n"x	Extended commands	+++
aRAW	76	ESC[n"r	Next n chars are raw	+++

Legend:

- ISO indicates that the sequence has been defined by the International Standards Organization. This is also very similar to ANSI x3.64.
- DEC indicates a control sequence defined by Digital Equipment Corporation.
- +++ indicates a sequence unique to Amiga.
- n stands for a decimal number expressed as a set of ASCII digits. In the aRAW string ESC[5"rHELLO, n is substituted by 5, the number of RAW characters you send to the printer.

ISO Color Table

0	Black
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White
8	NC
9	Default

Suggested Typefaces

0	Default typeface
1	Line Printer or equivalent
2	Pica or equivalent
3	Elite or equivalent
4	Helvetica or equivalent
5	Times Roman or equivalent
6	Gothic or equivalent
7	Script or equivalent
8	Prestige or equivalent
9	Caslon or equivalent

10 Orator or equivalent

1.14 10 Printer Device / Obtaining Printer Specific Data

Information about the printer in use can be obtained by reading the PrinterData and PrinterExtendedData structures. The values found in these structures are determined by the printer driver selected through Preferences. The data structures are defined in devices/prtbase.h.

Printer specific data is returned in printerIO when the printer device is opened. To read the structures, you must first set the PrinterData structure to point to iodrp.io_Device of the printerIO used to open the device and then set PrinterExtendedData to point to the extended data portion of PrinterData.

Printer_Data.c

1.15 10 Printer Device / Reading and Changing Printer Preferences Settings

The user preferences can be read and changed without running the Workbench Preferences tool. Reading printer preferences can be done by referring to PD->pd_Preferences. Listed on the next page are the printer Preferences fields and their valid ranges.

Text Preferences

PrintPitch	- PICA, ELITE, FINE
PrintQuality	- DRAFT, LETTER
PrintSpacing	- SIX_LPI, EIGHT_LPI
PrintLeftMargin	- 1 to PrintRightMargin
PrintRightMargin	- PrintLeftMargin to 999
PaperLength	- 1 to 999
PaperSize	- US_LETTER, US_LEGAL, N_TRACTOR, W_TRACTOR, CUSTOM
PaperType	- FANFOLD, SINGLE

Graphic Preferences

PrintImage	- IMAGE_POSITIVE, IMAGE_NEGATIVE
PrintAspect	- ASPECT_HORIZ, ASPECT_VERT
PrintShade	- SHADE_BW, SHADE_GREYSCALE, SHADE_COLOR
PrintThreshold	- 1 to 15
PrintFlags	- CORRECT_RED, CORRECT_GREEN, CORRECT_BLUE, CENTER_IMAGE, IGNORE_DIMENSIONS, BOUNDED_DIMENSIONS, ABSOLUTE_DIMENSIONS, PIXEL_DIMENSIONS, MULTIPLY_DIMENSIONS, INTEGER_SCALING, ORDERED_DITHERING, HALFTONE_DITHERING, FLOYD_DITHERING, ANTI_ALIAS, GREY_SCALE2
PrintMaxWidth	- 0 to 65535
PrintMaxHeight	- 0 to 65535
PrintDensity	- 1 to 7
PrintXOffset	- 0 to 255

This example program changes various settings in the printer device's copy of preferences.

```
Set_Prefs.c
```

```
Do Your Duty.
```

```
-----
```

The application program is responsible for range checking if the user is able to change the preferences from within the application.

1.16 10 Printer Device / Querying the Printer Device

The status of the printer port and registers can be determined by querying the printer device. The information returned will vary depending on the type of printer - parallel or serial - selected by the user. If parallel, the data returned will reflect the current state of the parallel port; if serial, the data returned will reflect the current state of the serial port.

You query the printer device by passing an IOStdReq to the device with PRD_QUERY set in io_Command and a pointer to a structure to hold the status set in io_Data.

```
struct PStat
{
    UBYTE LSB;          /* least significant byte of status */
    UBYTE MSB;          /* most significant byte of status */
};

union printerIO *PrintIO;

struct PStat status;

PrintIO->ios.io_Data = &status;      /* point to status structure */
PrintIO->ios.io_Command = PRD_QUERY;
DoIO((struct IORequest *)request);
```

The status is returned in the two UBYTES set in the io_Data field. The printer type, either serial or parallel, is returned in the io_Actual field.

io_Data	Bit	Active	Function (Serial Device)
-----	---	-----	-----
LSB	0	low	reserved
	1	low	reserved
	2	low	reserved
	3	low	Data Set Ready
	4	low	Clear To Send
	5	low	Carrier Detect
	6	low	Ready To Send
	7	low	Data Terminal Ready
MSB	8	high	read buffer overflow

	9	high	break sent (most recent output)
	10	high	break received (as latest input)
	11	high	transmit x-OFFed
	12	high	receive x-OFFed
	13-15	high	reserved
io_Data	Bit	Active	Function (Parallel Device)
-----	---	-----	-----
LSB	0	high	printer busy (offline)
	1	high	paper out
	2	high	printer selected
	3	-	read=0; write=1
	4-7		reserved
MSB	8-15		reserved
io_Actual			1-parallel, 2-serial

1.17 10 Printer Device / Error Codes from the Printer Device

The printer device returns error codes whenever an operation is attempted. There are two types of error codes that can be returned. Printer device error codes have positive values; Exec I/O error codes have negative values. Therefore, an application should check for a non-zero return code as evidence of an error, not simply a value greater than zero.

```
PrintIO->ios.io_Length  = strlen(outbuffer);
PrintIO->ios.io_Data    = (APTR)outbuffer;
PrintIO->ios.io_Command = PRD_RAWWRITE;
if (DoIO((struct IORequest *)PrintIO))
    printf("RAW Write failed. Error: %d ",PrintIO->ios.io_Error);
```

The error is found in io_Error.

PRINTER DEVICE ERROR CODES

Error	Value	Explanation
-----	-----	-----
PDERR_NOERR	0	Operation successful
PDERR_CANCEL	1	User canceled request
PDERR_NOTGRAPHICS	2	Printer cannot output graphics
PDERR_INVERTHAM	3	OBSOLETE
PDERR_BADDIMENSION	4	Print dimensions are illegal
PDERR_DIMENSIONOVERFLOW	5	OBSOLETE
PDERR_INTERNALMEMORY	6	No memory available for internal variables
PDERR_BUFFERMEMORY	7	No memory available for print buffer

EXEC ERROR CODES

Error	Value	Explanation
----	-----	-----
IOERR_OPENFAIL	-1	Device failed to open
IOERR_ABORTED	-2	Request terminated early (after AbortIO())
IOERR_NOCMD	-3	Command not supported by device
IOERR_BADLENGTH	-4	Not a valid length

1.18 10 Printer Device / Dumping a Rastport to a Printer

You dump a RastPort (drawing area) to a graphics capable printer by passing an IODRPreq to the device with PRD_DUMPRPORT set in io_Command along with several parameters that define how the dump is to be rendered.

```
union printerIO *PrintIO
struct RastPort *rastPort;
struct ColorMap *colorMap;
ULONG modeid;
UWORD sx, sy, sw, sh;
LONG dc, dr;
UWORD s;

PrintIO->iodrp.io_RastPort = rastPort; /* pointer to RastPort */
PrintIO->iodrp.io_ColorMap = colorMap; /* pointer to color map */
PrintIO->iodrp.io_Modes = modeid;      /* ModeID of ViewPort */
PrintIO->iodrp.io_SrcX = sx;           /* RastPort X offset */
PrintIO->iodrp.io_SrcY = sy;           /* RastPort Y offset */
PrintIO->iodrp.io_SrcWidth = sw;       /* print width from X offset */
PrintIO->iodrp.io_SrcHeight = sh;      /* print height from Y offset */
PrintIO->iodrp.io_DestCols = dc;       /* pixel width */
PrintIO->iodrp.io_DestRows = dr;       /* pixel height */
PrintIO->iodrp.io_Special = s;         /* flags */
PrintIO->iodrp.io_Command = PRD_DUMPRPORT;
SendIO((struct IORequest *)request);
```

The asynchronous SendIO() routine is used in this example instead of the synchronous DoIO(). A call to DoIO() does not return until the I/O request is finished. A call to SendIO() returns immediately. This allows your task to do other processing such as checking if the user wants to abort the I/O request. It should also be used when writing a lot of text or raw data with CMD_WRITE and PRD_RAWWRITE.

Here is an overview of the possible arguments for the RastPort dump.

io_RastPort	A pointer to a RastPort. The RastPort's bitmap could be in Fast memory.
io_ColorMap	A pointer to a ColorMap. This could be a custom one.
io_Modes	The viewmode flags or the ModeID returned from GetVPMODEID() (V36).
io_SrcX	X offset in the RastPort to start printing from.
io_SrcY	Y offset in the RastPort to start printing from.
io_SrcWidth	Width of the RastPort to print from io_SrcX.
io_SrcHeight	Height of the RastPort to print from io_SrcY.
io_DestCols	Width of the dump in printer pixels.
io_DestRows	Height of the dump in printer pixels.
io_Special	Flag bits (described below).

Looking at these arguments you can see the enormous flexibility the printer device offers for dumping a RastPort. The RastPort pointed to could be totally custom defined. This flexibility means it is possible to build a BitMap with the resolution of the printer. This would result in having one pixel of the BitMap correspond to one pixel of the printer. In

other words, only the resolution of the output device would limit the final result. With 12 bit planes and a custom ColorMap, you could dump 4096 colors - without the HAM limitation - to a suitable printer. The offset, width and height parameters allow dumps of any desired part of the picture. Finally the ViewPort mode, `io_DestCols`, `io_DestRows` parameters, together with the `io_Special` flags define how the dump will appear on paper and aid in getting the correct aspect ratio.

Printer Special Flags

Printing With Corrected Aspect Ratio

Strip Printing

Additional Notes About Graphic Dumps

1.19 10 / Dumping a Rastport to a Printer / Printer Special Flags

The printer special flags (`io_Flags`) of the `IODRPreq` provide a high degree of control over the printing of a RastPort.

<code>SPECIAL_ASPECT</code>	Allows one of the dimensions to be reduced/expanded to preserve the correct aspect ratio of the printout.
<code>SPECIAL_CENTER</code>	Centers the image between the left and right edge of the paper.
<code>SPECIAL_NOFORMFEED</code>	Prevents the page from being ejected after a graphics dump. Usually used to mix graphics and text or multiple graphics dump on a page oriented printer (normally a laser printer).
<code>SPECIAL_NOPRINT</code>	The print size will be computed, and set in <code>io_DestCols</code> and <code>io_DestRows</code> , but won't print. This way the application can see what the actual printsize in printerpixels would be.
<code>SPECIAL_TRUSTME</code>	Instructs the printer not to send a reset before and after the dump. This flag is obsolete for V1.3 (and higher) drivers.
<code>SPECIAL_DENSITY1-7</code>	This flag bit is set by the user in Preferences. Refer to "Reading and Changing the Printer Preferences Settings" if you want to change to density of the printout. (Or any other setting for that matter.)
<code>SPECIAL_FULLCOLS</code>	The width is set to the maximum possible, as determined by the printer or the configuration limits.
<code>SPECIAL_FULLROWS</code>	The height is set to the maximum possible, as determined by the printer or the configuration limits.
<code>SPECIAL_FRACCOLS</code>	Informs the printer device that the value in <code>io_DestCols</code> is to be taken as a longword binary

fraction of the maximum for the dimension. For example, if `io_DestCols` is 0x8000, the width would be 1/2 (0x8000 / 0xffff) of the width of the paper.

<code>SPECIAL_FRACROWS</code>	<p> Informs the printer device that the value in <code>io_DestRows</code> is to be taken as a longword binary fraction for the dimension. </p>
<code>SPECIAL_MILCOLS</code>	<p> Informs the printer device that the value in <code>io_DestCols</code> is specified in thousandths of an inch. For example, if <code>io_DestCols</code> is 8000, the width of the printout would be 8.000 inches. </p>
<code>SPECIAL_MILROWS</code>	<p> Informs the printer device that the value in <code>io_DestRows</code> is specified in thousandths of an inch. </p>

The flags are defined in the include file `devices/printer.h`.

1.20 10 / Dumping Rastport to a Printer / Printing Corrected Aspect Ratio

Using the special flags it is fairly easy to ensure a graphic dump will have the correct aspect ratio on paper. There are some considerations though when printing a non-displayed RastPort. One way to get a corrected aspect ratio dump is to calculate the printer's ratio from `XDotsInch` and `YDotsInch` (taking into account that the printer may not have square pixels) and then adjust the width and height parameters accordingly. You then ask for a non-aspect-ratio-corrected dump since you already corrected it yourself.

Another possibility is having the printer device do it for you. To get a correct calculation you could build your RastPort dimensions in two ways:

1. Using an integer multiple of one of the standard (NTSC) display resolutions and setting the `io_Modes` argument accordingly. For example if your RastPort dimensions were 1280 x 800 (an even multiple of 640 x 400) you would set `io_Modes` to `LACE | HIRES`. Setting the `SPECIAL_ASPECT` flag would enable the printer device to properly calculate the aspect ratio of the image.
2. When using an arbitrary sized RastPort, you can supply the `ModeID` of a display mode which has the aspect ratio you would like for your RastPort. The aspect ratio of the various display modes are defined as ticks-per-pixel in the `Resolution` field of the `DisplayInfo` structure. You can obtain this value from the graphics database. For example, the resolution of Productivity Mode is 22:22, in other words, 1:1, perfect for a RastPort sized to the limits of the output device. See the "Graphics Library" chapters of the Amiga ROM Kernel Reference Manual: Libraries for general information on the graphics system.

The following example will dump a RastPort to the printer and wait for either the printer to finish or the user to cancel the dump and act accordingly.

Demo_Dump.c

1.21 10 / Dumping a Rastport to a Printer / Strip Printing

Strip printing is a method which allows you to print a picture that normally requires a large print buffer when there is not much memory available. This would allow, for example, a RastPort to be printed at a higher resolution than it was drawn in. Strip printing is done by creating a temporary RastPort as wide as the source RastPort, but not as high. The source RastPort is then rendered, a strip at a time, into the temporary RastPort which is dumped to the printer.

The height of the strip to dump must be an integer multiple of the printer's NumRows if a non-aspect-ratio-corrected image is to be printed.

For an aspect-ratio-corrected image, the SPECIAL_NOPRINT flag will have to be used to find an io_DestRows that is an integer multiple of NumRows. This can be done by varying the source height and asking for a SPECIAL_NOPRINT dump until io_DestRows holds a number that is an integer multiple of the printer's NumRows.

If smoothing is to work with strip printing, a raster line above and below the actual area should be added. The line above should be the last line from the previous strip, the line below should be the first line of the next strip. Of course, the first strip should not have a line added above and the last strip should not have a line added below.

The following is a strip printing procedure for a RastPort which is 200 lines high.

First strip

- * copy source line 0 through 50 (51 lines) to strip RastPort lines 0 through 50 (51 lines).
- * io_SrcY = 0, io_Height = 50.
- * the printer device can see there is no line above the first line to dump (since SrcY = 0) and that there is a line below the last line to dump (since there is a 51 line RastPort and only 50 lines are dumped).

Second strip

- * copy source line 49 through 100 (52 lines) to strip RastPort lines 0 through 51 (52 lines).
- * io_SrcY = 1, io_Height = 50.
- * the printer device can see there is a line above the first line to dump (since SrcY = 1) and that there is a line below the last line to dump (since there is a 52 line RastPort and only 50 lines are dumped).

Third strip

- * copy source line 99 through 150 (52 lines) to strip RastPort lines 0 through 51 (52 lines).
 - * io_SrcY = 1, io_Height = 50.
 - * the printer device can see there is a line above the first line to
-

dump (since SrcY = 1) and that there is a line below the last line to
dump (since there is a 52 line RastPort and only 50 lines are dumped).

Fourth strip

- * copy source line 149 through 199 (51 lines) to strip RastPort lines 0 through 50 (51 lines).
- * io_SrcY = 1, io_Height = 50.
- * the printer device can see there is a line above the first line to dump (since SrcY = 1) and that there is no line below the last line to dump (since there is a 51 line RastPort and only 50 lines are dumped).

1.22 10 / Dumping Rastport to a Printer / Additional Notes on Graphic Dumps

1. When dumping a 1 bitplane image select the black and white mode in Preferences. This is much faster than a grey-scale or color dump.
2. Horizontal dumps are much faster than vertical dumps.
3. Smoothing doubles the print time. Use it for final copy only.
4. F-S dithering doubles the print time. Ordered and half-tone dithering incur no extra overhead.
5. The lower the density, the faster the printout.
6. Friction-fed paper tends to be much more accurate than tractor-fed paper in terms of vertical dot placement (i.e., less horizontal strips or white lines).
7. Densities which use more than one pass tend to produce muddy grey-scale or color printouts. It is recommended not to choose these densities when doing a grey-scale or color dump.

Keep This in Mind.

It is possible that the printer has been instructed to receive a certain amount of data and is still in an "expecting" state if an I/O request has been aborted by the user. This means the printer would try to finish the job with the data the next I/O request might send. Currently the best way to overcome this problem is for the printer to be reset.

1.23 10 Printer Device / Creating a Printer Driver

Creating the printer-dependent modules for the printer device involves writing the data structures and code, compiling and assembling them, and linking to produce an Amiga binary object file. The modules a driver contains varies depending on whether the printer is non-graphics or graphics capable.

All drivers contain these modules:

macros.i	- include file for init.asm, contains printer device macro definitions
printertag.asm	- printer specific capabilities such as density, character sets and color
init.asm	- opens the various libraries required by the printer driver. This will be the same for all printers
data.c	- contains printer device RAW commands and the extended character set supported by the printer
dospecial.c	- printer specific special processing required for printer device commands like aSLRM and aSFC

Graphic printer drivers require these additional modules:

render.c	- printer specific processing to do graphics output and fill the output buffer
transfer.c	- printer specific processing called by render.c to output the buffer to the printer. Code it in assembly if speed is important
density.c	- printer specific processing to construct the proper print density commands

The first piece of the printer driver is the PrinterSegment structure described in devices/prtbase.h (this is pointed to by the BPTR returned by the LoadSeg() of the object file). The PrinterSegment contains the PrinterExtendedData (PED) structures (also described in devices/prtbase.h) at the beginning of the object. The PED structure contains data describing the capabilities of the printer, as well as pointers to code and other data. Here is the assembly code for a sample PrinterSegment, which would be linked to the beginning of the sequence of files as printertag.asm.

printertag.asm

The printer name should be the brand name of the printer that is available for use by programs wishing to be specific about the printer name in any diagnostic or instruction messages. The four functions at the top of the structure are used to initialize this printer-dependent code:

```
(* (PED->ped_Init)) (PD) ;
    This is called when the printer-dependent code is loaded and provides
    a pointer to the printer device for use by the printer-dependent
    code. It can also be used to open up any libraries or devices needed
    by the printer-dependent code.

(* (PED->ped_Expunge)) () ;
    This is called immediately before the printer-dependent code is
    unloaded, to allow it to close any resources obtained at
    initialization time.

(* (PED->ped_Open)) (ior) ;
    This is called in the process of an OpenDevice() call, after the
    Preferences are read and the correct primitive I/O device (parallel
    or serial) is opened. It must return zero if the open is successful,
    or non-zero to terminate the open and return an error to the user.

(* (PED->ped_Close)) (ior) ;
```

This is called in the process of a `CloseDevice()` call to allow the printer-dependent code to close any resources obtained at open time.

The `pd_` variable provided as a parameter to the initialization call is a pointer to the `PrinterData` structure described in `devices/prtbase.h`. This is also the same as the `io_Device` entry in printer I/O requests.

`pd_SegmentData`

This points back to the `PrinterSegment`, which contains the PED.

`pd_PrintBuf`

This is available for use by the printer-dependent code - it is not otherwise used by the printer device.

`(*pd_PWrite)(data, length);`

This is the interface routine to the primitive I/O device. This routine uses two I/O requests to the primitive device, so writes are double-buffered. The data parameter points to the byte data to send, and the length is the number of bytes.

`(*pd_PBothReady)();`

This waits for both primitive I/O requests to complete. This is useful if your code does not want to use double buffering. If you want to use the same data buffer for successive `pd_PWrites`, you must separate them with a call to this routine.

`pd_Preferences`

This is the copy of Preferences in use by the printer device, obtained when the printer was opened.

The timeout field is the number of seconds that an I/O request from the printer device to the primitive I/O device (parallel or serial) will remain posted and unsatisfied before the timeout requester is presented to the user. The timeout value should be long enough to avoid the requester during normal printing.

The `PrintMode` field is a flag which indicates whether text has been printed or not (1 means printed, 0 means not printed). This flag is used in drivers for page oriented printers to indicate that there is no alphanumeric data waiting for a formfeed.

Writing Alphanumeric Printer Drivers

Writing A Graphics Printer Driver

Testing The Printer Driver

1.24 10 / Creating a Printer Driver / Writing Alphanumeric Printer Drivers

The alphanumeric portion of the printer driver is designed to convert ANSI x3.64 style commands into the specific escape codes required by each individual printer. For example, the ANSI code for underline-on is `ESC[4m`. The Commodore MPS-1250 printer would like a `ESC[-1` to set underline-on. The HP LaserJet accepts `ESC[&dD` as a start underline command. By using the printer driver, all printers may be handled in a similar manner.

There are two parts to the alphanumeric portion of the printer driver: the Command Table data table and the DoSpecial() routine.

```
Command Table
DoSpecial()
Printertag.asm
Extended Character Table
Character Conversion Routine
```

1.25 10 // Writing An Alphanumeric Printer Driver / Command Table

The CommandTable is used to convert all escape codes that can be handled by simple substitution. It has one entry per ANSI command supported by the printer driver. When you are creating a custom CommandTable, you must maintain the order of the commands in the same sequence as that shown in devices/printer.h. By placing the specific codes for your printer in the proper positions, the conversion takes place automatically.

Octal knows NULL.

If the code for your printer requires a decimal 0 (an ASCII NULL character), you enter this NULL into the CommandTable as octal 376 (decimal 254).

Placing an octal value of 377 (255 decimal) in a position in the command table indicates to the printer device that no simple conversion is available on this printer for this ANSI command. For example, if a daisy-wheel printer does not have a foreign character set, 377 octal (255 decimal) is placed in that position in the command table. However, 377 in a position can also mean that the ANSI command is to be handled by code located in the DoSpecial() function. For future compatibility all printer commands should be present in the command table, and those not supported by the printer filled with the dummy entry 377 octal.

1.26 10 // Writing An Alphanumeric Printer Driver / DoSpecial()

The DoSpecial() function is meant to implement all the ANSI functions that cannot be done by simple substitution, but can be handled by a more complex sequence of control characters sent to the printer. These are functions that need parameter conversion, read values from Preferences, and so on. Complete routines can also be placed in dospecial.c. For instance, in a driver for a page oriented-printer such as the HP LaserJet, the dummy Close() routine from the init.asm file would be replaced by a real Close() routine in dospecial.c. This close routine would handle ejecting the paper after text has been sent to the printer and the printer has been closed.

The DoSpecial() function is set up as follows:

```
#include "exec/types.h"
#include "devices/printer.h"
#include "devices/prtbase.h"
```

```
extern struct PrinterData *PD;

DoSpecial(command,outputBuffer,vline,currentVMI,crlfFlag,Parms)
UBYTE outputBuffer[];
UWORD *command;
BYTE *vline;
BYTE *currentVMI;
BYTE *crlfFlag;
UBYTE Parms[];
{
    /* code begins here... */

where

command
    points to the command number. The devices/printer.h file contains the
    definitions for the routines to use (aRIN is initialize, and so on).

vline
    points to the value for the current line position.

currentVMI
    points to the value for the current line spacing.

crlfFlag
    points to the setting of the "add line feed after carriage return"
    flag.

Parms
    contain whatever parameters were given with the ANSI command.

outputBuffer
    points to the memory buffer into which the converted command is
    returned.
```

Almost every printer will require an aRIN (initialize) command in DoSpecial(). This command reads the printer settings from Preferences and creates the proper control sequence for the specific printer. It also returns the character set to normal (not italicized, not bold, and so on). Other functions depend on the printer.

Certain functions are implemented both in the CommandTable and in the DoSpecial() routine. These are functions such as superscript, subscript, PLU (partial line up), and PLD (partial line down), which can often be handled by a simple conversion. However, some of these functions must also adjust the printer device's line-position variable.

Save the Data!

Some printers lose data when sent their own reset command. For this reason, it is recommended that if the printer's own reset command is going to be used, PD->pd_PWaitEnabled should be defined to be a character that the printer will not print. This character should be put in the reset string before and after the reset character(s) in the command table.

In the EpsonX[CBM_MPS-1250] DoSpecial() function you'll see

```

if (*command == aRIS)
{
    /* reset command */
    PD->pd_PWaitEnabled = \375; /* preserve that data! */
}

```

while in the command table the string for reset is defined as "`\\375\\033@\\375`". This means that when the printer device outputs the reset string "`\\033@`", it will first see the "`\\375`", wait a second and output the reset string. While the printer is resetting, the printer device gets the second "`\\375`" and waits another second. This ensures that no data will be lost if a reset command is embedded in a string.

1.27 10 // Writing An Alphanumeric Printer Driver / Printertag.asm

For an alphanumeric printer the printer-specific values that need to be filled in `printertag.asm` are as follows:

```

MaxColumns
    the maximum number of columns the printer can print across the page.

NumCharSets
    the number of character sets which can be selected.

8BitChars
    a pointer to an extended character table. If the field is null, the
    default table will be used.

ConvFunc
    a pointer to a character conversion routine. If the field is null, no
    conversion routine will be used.

```

1.28 10 // Writing An Alphanumeric Printer Driver / Extended Character Table

The `8BitChars` field could contain a pointer to a table of characters for the ASCII codes `$A0` to `$FF`. The symbols for these codes are shown in the IFF Appendix of this manual. If this field contains a `NULL`, it means no specific table is provided for the driver, and the default table is to be used instead.

Care should be taken when generating this table because of the way the table is parsed by the printer device. Valid expressions in the table include `\011` where `011` is an octal number, `\\000` for null and `\\n` where `n` is a 1 to 3 digit decimal number. To enter an actual backslash in the table requires the somewhat awkward `\\`. As an example, here is a list of the first entries of the `EpsonxX[CBM_MPS-1250]` table:

```

char *ExtendedCharTable[] =
{
    " ",                /* NBSB */
    "\\033R\\007[\\033R\\0", /* i */
    "c\\010|",          /* c | */

```

```

"\033R\003#\033R\0",      /* L- */
"\033R\005$\033R\0",      /* o */
"\033R\010\\\\\033R\0",    /* Y- */
"|",                        /* | */
"\033R\002@\033R\0",      /* SS */
"\033R\001~\033R\0",      /* " */
"c",                        /* copyright */
"\033S\0a\010_\033T",     /* a_ */
"<",                        /* << */
"~",                        /* - */
"-",                        /* SHY */
"r",                        /* registered trademark */
"-",                        /* - */
/* more entries go here */
};

```

1.29 10 // Writing An Alphanumeric Printer Driver / Character Conversion

The ConvFunc field contains a pointer to a character conversion function that allows you to selectively translate any character to a combination of other characters. If no translation conversion is necessary (for most printers it isn't), the field should contain a null.

ConvFunc() arguments are a pointer to a buffer, the character currently processed, and a CR/LF flag. The ConvFunc() function should return a -1 if no conversion has been done. If the character is not to be added to the buffer, a 0 can be returned. If any translation is done, the number of characters added to the buffer must be returned.

Besides simple character translation, the ConvFunc() function can be used to add features like underlining to a printer which doesn't support them automatically. A global flag could be introduced that could be set or cleared by the DoSpecial() function. Depending on the status of the flag the ConvFunc() routine could, for example, put the character, a backspace and an underline character in the buffer and return 3, the number of characters added to the buffer.

The ConvFunc() function for this could look like the following example:

```

#define DO_UNDERLINE    0x01
#define DO_BOLD         0x02
/* etc */

external short myflags;

int ConvFunc(buffer, c, crlf_flag)
char *buffer, c;
int crlf_flag
{
    int nr_of_chars_added = 0;

    /* for this example we only do this for chars in the 0x20-0x7e range */
    /* Conversion of ESC (0x1b) and CSI (0x9b) is NOT recommended */

    if (c > 0x1f && c < 0x7f)

```

```

{
    /* within space - ~ range ? */
    if (myflags & DO_UNDERLINE)
    {
        *buffer++ = c;          /* the character itself */
        *buffer++ = 0x08;       /* a backspace */
        *buffer++ = '_';        /* an underline char */
        nr_of_chars_added = 3;  /* added three chars to buffer */
    }
    if (myflags & DO_BOLD)
    {
        if (nr_of_chars_added)
        {
            /* already have added something */
            *buffer++ = 0x08;    /* so we start with backspace */
            ++nr_of_chars_added; /* and increment the counter */
        }
        *buffer++ = c;
        *buffer++ = 0x08;
        *buffer++ = c;
        ++nr_of_chars_added;
        if (myflags & DO_UNDERLINE)
        {
            /* did we do underline too? */
            *buffer++ = 0x08;    /* then backspace again */
            *buffer++ = '_';     /* (printer goes crazy by now) */
            nr_of_chars_added += 2; /* two more chars */
        }
    }
}
if (nr_of_chars_added)
    return(nr_of_chars_added); /* total nr of chars we added */
else
    return(-1);                /* we didn't do anything */
}

```

In DoSpecial() the flagbits could be set or cleared, with code like the following:

```

if (*command == aRIS)          /* reset command */
    myflags = 0;               /* clear all flags */

if (*command == aRIN)          /* initialize command */
    myflags = 0;

if (*command == aSGR0)         /* 'PLAIN' command */
    myflags = 0;

if (*command == aSGR4)         /* underline on */
    myflags |= DO_UNDERLINE;   /* set underline bit */

if (*command == aSGR24)        /* underline off */
    myflags &= ~DO_UNDERLINE;  /* clear underline bit */

if (*command == aSGR1)         /* bold on */
    myflags |= DO_BOLD;        /* set bold bit */

if (*command == aSGR22)        /* bold off */
    myflags &= ~DO_BOLD;       /* clear bold bit */

```

Try to keep the expansions to a minimum so that the throughput will not be slowed down too much, and to reduce the possibility of data overrunning the printer device buffer.

1.30 10 / Creating a Printer Driver / Writing A Graphics Printer Driver

Designing the graphics portion of a custom printer driver consists of two steps: writing the printer-specific `Render()`, `Transfer()` and `SetDensity()` functions, and replacing the printer-specific values in `printertag.asm`. `Render()`, `Transfer()` and `SetDensity()` comprise `render.c`, `transfer.c`, and `density.c` modules, respectively.

A printer that does not support graphics has a very simple form of `Render()`; it returns an error. Here is sample code for `Render()` for a non-graphics printer (such as an Alphacom or Diablo 630):

```
#include "exec/types.h"
#include "devices/printer.h"
int Render()
{
    return(PDERR_NOTGRAPHICS);
}
```

The following section describes the contents of a typical driver for a printer that does support graphics.

`Render()` `Transfer()` `SetDensity()` `Printertag.asm`

1.31 10 / Writing A Graphics Printer Driver / `Render()`

This function is the main printer-specific code module and consists of seven parts referred to here as cases:

- * Pre-Master initialization (Case 5)
- * Master initialization (Case 0)
- * Putting the pixels in a buffer (Case 1)
- * Dumping a pixel buffer to the printer (Case 2)
- * Closing down (Case 4)
- * Clearing and initializing the pixel buffer (Case 3)
- * Switching to the next color(Case 6) (special case for multi-color printers)

State Your Case.

The numbering of the cases reflects the value of each step as a case in a C-language switch statement. It does not denote the order that

the functions are executed; the order in which they are listed above denotes that.

For each case, Render() receives four long variables as parameters: ct, x, y and status. These parameters are described below for each of the seven cases that Render() must handle.

Pre-Master initialization (Case 5)

Parameters:

```
ct - 0 or pointer to the IODRPRReq structure passed to PCDumpRPort
x  - io_Special flag from the IODRPRReq structure
y  - 0
```

When the printer device is first opened, Render() is called with ct set to 0, to give the driver a chance to set up the density values before the actual graphic dump is called.

The parameter passed in x will be the io_Special flag which contains the density and other SPECIAL flags. The only flags used at this point are the DENSITY flags, all others should be ignored. Never call PWrite() during this case. When you are finished handling this case, return PDERR_NOERR.

Master initialization (Case 0).

Parameters:

```
ct - pointer to a IODRPRReq structure
x  - width (in pixels) of printed picture
y  - height (in pixels) of printed picture
```

Everything is A-OK.

At this point the printer device has already checked that the values are within range for the printer. This is done by checking values listed in printertag.asm.

The x and y value should be used to allocate enough memory for a command and data buffer for the printer. If the allocation fails, PDERR_BUFFERMEMORY should be returned. In general, the buffer needs to be large enough for the commands and data required for one pass of the print head. These typically take the following form:

```
<start gfx cmd> <data> <end gfx cmd>
```

The <start gfx cmd> should contain any special, one-time initializations that the printer might require such as:

- * Carriage Return - some printers start printing graphics without returning the printhead. Sending a CR assures that printing will start from the left edge.
- * Unidirectional - some printers which have a bidirectional mode produce non-matching vertical lines during a graphics dump, giving a wavy result. To prevent this, your driver should set the printer to unidirectional mode.

- * Clear margins - some printers force graphic dumps to be done within the text margins, thus they should be cleared.
- * Other commands - enter the graphics mode, set density, etc.

Multi-Pass? Don't Forget the Memory.

In addition to the memory for commands and data, a multi-pass color printer must allocate enough buffer space for each of the different color passes.

The printer should never be reset during the master initialization case. This will cause problems during multiple dumps. Also, the pointer to the IODRPreq structure in ct should not be used except for those rare printers which require it to do the dump themselves. Return the PDERR_TOOKCONTROL error in that case so that the printer device can exit gracefully.

PDERR_TOOKCONTROL, An Error in Name Only.

The printer device error code, PDERR_TOOKCONTROL, is not an error at all, but an internal indicator that the printer driver is doing the graphic dump entirely on its own. The printer device can assume the dump has been done. The calling application will not be informed of this, but will receive PDERR_NOERR instead.

The example render.c functions listed at the end of this chapter use double buffering to reduce the dump time which is why the AllocMem() calls are for BUFSIZE times two, where BUFSIZE represents the amount of memory for one entire print cycle. However, contrary to the example source code, allocating the two buffers independently of each other is recommended. A request for one large block of contiguous memory might be refused. Two smaller requests are more likely to be granted.

Putting the pixels in a buffer (Case 1).

Parameters:

ct - pointer to a PrtInfo structure.
 x - PCM color code (if the printer is PCC_MULTI_PASS).
 y - printer row # (the range is 0 to pixel height - 1).

In this case, you are passed an entire row of YMCKB intensity values (Yellow, Magenta, Cyan, Black). To handle this case, you call the Transfer() function in the transfer.c module. You should return PDERR_NOERR after handling this case. The PCM-defines for the x parameter from the file devices/prtgfx.h are PCMYELLOW, PCMMAGENTA, PCMCYAN and PCMBLACK.

Dumping a pixel buffer to the printer (Case 2).

Parameters:

ct - 0
 x - 0
 y - # of rows sent (the range is 1 to NumRows).

At this point the data can be Run Length Encoded (RLE) if your printer supports it. If the printer doesn't support RLE, the data should be white-space stripped. This involves scanning the buffer from end to beginning for the position of the first occurrence of a non-zero value. Only the data from the beginning of the buffer to this position should be sent to the printer. This will significantly reduce print times.

The value of y can be used to advance the paper the appropriate number of pixel lines if your printer supports that feature. This helps prevent white lines from appearing between graphic dumps.

You can also do post-processing on the buffer at this point. For example, if your printer uses the hexadecimal number \$03 as a command and requires the sequence \$03 \$03 to send \$03 as data, you would probably want to scan the buffer and expand any \$03s to \$03 \$03 during this case. Of course, you'll need to allocate space somewhere in order to expand the buffer.

The error from PWrite() should be returned after this call.

Clearing and initializing the pixel buffer (Case 3)

Parameters:

ct - 0
x - 0
y - 0

The printer driver does not send blank pixels so you must initialize the buffer to the value your printer uses for blank pixels (usually 0). Clearing the buffer should be the same for all printers. Initializing the buffer is printer specific, and it includes placing the printer-specific control codes in the buffer before and after the data.

This call is made before each Case 2 call. Clear your active print buffer - remember you are double buffering - and initialize it if necessary. After this call, PDERR_NOERR should be returned.

Closing Down (Case 4).

Parameters:

ct - error code
x - io_Special flag from the IODRPreq structure
y - 0

This call is made at the end of the graphic dump or if the graphic dump was cancelled for some reason. At this point you should free the printer buffer memory. You can determine if memory was allocated by checking that the value of PD->pd_PrintBuf is not NULL. If memory was allocated, you must wait for the print buffers to clear (by calling PBothReady) and then deallocate the memory. If the printer - usually a page oriented printer - requires a page eject command, it can be given here. Before you do, though, you should check the SPECIAL_NOFORMFEED bit in x. Don't issue the command if it is set.

If the error condition in ct is PDERR_CANCEL, you should not PWrite().

This error indicates that the user is trying to cancel the dump for whatever reason. Each additional PWrite() will generate another printer trouble requester. Obviously, this is not desirable.

During this render case PWrite() could be used to:

- * reset the line spacing. If the printer doesn't have an advance 'n' dots command, then you'll probably advance the paper by changing the line spacing. If you do, set it back to either 6 or 8 lpi (depending on Preferences) when you are finished printing.
- * set bidirectional mode if you selected unidirectional mode in render Case 0.
- * set black text; some printers print the text in the last color used, even if it was in graphics mode.
- * restore the margins if you cancelled the margins in render Case 0.
- * any other command needed to exit the graphics mode, eject the page, etc.

Either PDERR_NOERR or the error from PWrite() should be returned after this call.

Switching to the next color (Case 6)

This call provides support for printers which require that colors be sent in separate passes. When this call is made, you should instruct the printer to advance its color panel. This case is only needed for printers of the type PCC_MULTIPASS, such as the CalComp ColorMaster.

1.32 10 / Writing A Graphics Printer Driver / Transfer()

Transfer() dithers and renders an entire row of pixels passed to it by the Render() function. When Transfer() gets called, it is passed 5 parameters:

Parameters:

```
PInfo      - a pointer to a PInfo structure
y          - the row number
ptr        - a pointer to the buffer
colors     - a pointer to the color buffers
BufOffset  - the buffer offset for interleaved printing.
```

The dithering process of Transfer() might entail thresholding, grey-scale dithering, or color-dithering each destination pixel.

If PInfo->pi_threshold is non-zero, then the dither value is:

```
PInfo->pi_threshold ^15
```

If PInfo->pi_threshold is zero, then the dither value is computed

by:

```
*(PInfo->pi_dmatrix + ((y & 3) * 2) + (x & 3))
```

where x is initialized to PInfo->pi_xpos and is incremented for each of the destination pixels. Since the printer device uses a 4x4 dither matrix, you must calculate the dither value exactly as given above. Otherwise, your driver will be non-standard and the results will be unpredictable.

The Transfer() function renders by putting a pixel in the print buffer based on the dither value. If the intensity value for the pixel is greater than the dither value as computed above, then the pixel should be put in the print buffer. If it is less than, or equal to the dither value, it should be skipped to process the next pixel.

Printer Color Class -----	Type of Dithering -----	Rendering logic -----
PCC_BW	Thresholding	Test the black value against the threshold value to see if you should render a black pixel.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel.
	Color	NA
PCC_YMC	Thresholding	Test the black value against the threshold value to see if you should render a black pixel. Print yellow, magenta and cyan pixel to make black.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel. Print yellow, magenta and cyan pixel to make black.
	Color	Test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan.
PCC_YMCB	Thresholding	Test the black value against the threshold value to see if you should render a black pixel.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel.
	Color	Test the black value against the dither value to see if you should render a black pixel. If black is not rendered, then test the yellow value against the dither value to see if you should render a yellow

		pixel. Repeat this process for magenta and cyan. (See the EpsonX_transfer.c file)
PCC_YMC_BW	Thresholding	Test the black value against the threshold value to see if you should render a black pixel.
	Grey Scale	Test the black value against the dither value to see if you should render a black pixel.
	Color	Test the yellow value against the dither value to see if you should render a yellow pixel. Repeat this process for magenta and cyan.

In general, if black is rendered for a specific printer dot, then the YMC values should be ignored, since the combination of YMC is black. It is recommended that the printer buffer be constructed so that the order of colors printed is yellow, magenta, cyan and black, to prevent smudging and minimize color contamination on ribbon color printers.

The example transfer.c files are provided in C for demonstration only. Writing this module in assembler can cut the time needed for a graphic dump in half. The EpsonX transfer.asm file is an example of this.

1.33 10 / Writing A Graphics Printer Driver / SetDensity()

SetDensity() is a short function which implements multiple densities. It is called in the Pre-master initialization case of the Render() function. It is passed the density code in density_code. This is used to select the desired density or, if the user asked for a higher density than is supported, the maximum density available. SetDensity() should also handle narrow and wide tractor paper sizes.

Densities below 80 dpi should not be supported in SetDensity(), so that a minimum of 640 dots across for a standard 8.5x11-inch paper is guaranteed. This results in a 1:1 correspondence of dots on the printer to dots on the screen in standard screen sizes. The HP LaserJet is an exception to this rule. Its minimum density is 75 dpi because the original HP LaserJet had too little memory to output a full page at a higher density.

1.34 10 / Writing A Graphics Printer Driver / Printertag.asm

For a graphic printer the printer-specific values that need to be filled in in printertag.asm are as follows:

MaxXDots

The maximum number of dots the printer can print across the page.

MaxYDots

The maximum number of dots the printer can print down the page.

Generally, if the printer supports roll or form feed paper, this value should be 0 indicating that there is no limit. If the printer has a definite y dots maximum (as a laser printer does), this number should be entered here.

XDotsInch

The dot density in x (supplied by SetDensity(), if it exists).

YDotsInch

The dot density in y (supplied by SetDensity(), if it exists).

PrinterClass

The printer class of the printer.

PPC_BWALPHA	black&white alphanumeric, no graphics.
PPC_BWGFX	black&white (only) graphics.
PPC_COLORALPHA	color alphanumeric, no graphics.
PPC_COLORGFX	color (and maybe black&white) graphics.

ColorClass

The color class the printer falls into.

PCC_BW	Black&White only
PCC_YMC	Yellow Magenta Cyan only.
PCC_YMC_BW	Yellow Magenta Cyan or Black&White, but not both
PCC_YMCB	Yellow Magenta Cyan Black
PCC_WB	White&Black only, 0 is BLACK
PCC_BGR	Blue Green Red
PCC_BGR_WB	Blue Green Red or Black&White
PCC_BGRW	Blue Green Red White

NumRows

The number of pixel rows printed by one pass of the print head. This number is used by the non-printer-specific code to determine when to make a render Case 2 call to you. You have to keep this number in mind when determining how big a buffer you'll need to store one print cycle's worth of data.

1.35 10 / Creating a Printer Driver / Testing The Printer Driver

A printer driver should be thoroughly tested before it is released. Though labor intensive, the alphanumeric part of a driver can be easily tested. The graphics part is more difficult. Following are some recommendations on how to test this part.

Start with a black and white (threshold 8), grey scale and color dump of the same picture. The color dump should be in color, of course. The grey scale dump should be like the color dump, except it will consist of patterns of black dots. The black and white dump will have solid black and solid white areas.

Next, do a dump with the DestX and DestY dots set to an even multiple of the XDotsInch and YDotsInch for the printer. For example, if the printer has a resolution of 120 x 144 dpi, a 480 x 432 dump could be done. This should produce a printed picture which covers 4 x 3 inches on paper. If

the width of the picture is off, then the wrong value for XDotsInch has been put in printertag.asm. If the height of the picture is off, the wrong value for YDotsInch is in printertag.asm.

Do a color dump as wide as the printer can handle with the density set to 7.

Make sure that the printer doesn't force graphic dumps to be done within the text margins. This can easily be tested by setting the text margins to 30 and 50, the pitch to 10 cpi and then doing a graphic dump wider than 2 inches. The dump should be left justified and as wide as you instructed. If the dump starts at character position 30 and is cut off at position 50, the driver will have to be changed. These changes involve clearing the margins before the dump (Case 0) and restoring the margins after the dump (Case 4). An example of this is present in render.c source example listed at the end of this chapter.

The Invisible Setup.

Before the graphic dump, some text must be sent to the printer to have the printer device initialize the printer. The "text" sent does not have to contain any printable characters (i.e., you can send a carriage return or other control characters).

As a final test, construct an image with a white background that has objects in it surrounded by white space. Dump this as black and white, grey scale and color. This will test the white-space stripping or RLE, and the ability of the driver to handle null lines. The white data areas should be separated by at least as many lines of white space as the NumRows value in the printertag.asm file.

1.36 10 Printer Device / Example Printer Driver Source Code

As an aid in writing printer drivers, source code for two different classes of printers is supplied. Both drivers have been successfully generated with Lattice C 5.10 and Lattice Assembler 5.10. The example drivers are:

```
EpsonX      A YMCB, 8 pin, multi-density interleaved printer.
HP_Laserjet  A black&white, multi-density, page-oriented printer.
```

All printer drivers use the following include file macros.i for init.asm.

```
macros.i
```

```
EpsonX
HP_Laserjet
```

1.37 10 / Example Printer Driver Source Code / EpsonX

For the EpsonX driver, both the assembly and C version of Transfer() are supplied. In the Makefile the (faster) assembly version is used to

generate the driver. The EpsonX driver can be generated with the included Makefile.

Makefile	init.asm	transfer.asm
macros.i	data.c	transfer.c
printertag.asm	dospecial.c	density.c
rev.i	render.c	

1.38 10 / Example Printer Driver Source Code / HP_Laserjet

The driver for the HP_LaserJet can be generated with the following Makefile.

Makefile	init.asm	transfer.asm
macros.i	data.c	transfer.c
printertag.asm	dospecial.c	density.c
hp_rev.i	render.c	

1.39 10 Printer Device / Additional Information on the Printer Device

Additional programming information on the printer device can be found in the include files and the Autodocs for the printer device. Both are contained in the Amiga ROM Kernel Reference Manual: Includes and Autodocs.

```
Printer Device Information
-----
INCLUDES      devices/printer.h
               devices/printer.i
               devices/prtbase.h
               devices/prtbase.i
               devices/prtgfx.h
               devices/prtgfx.i

AUTODOCS      printer.doc
```

Additional printer drivers can be found on Fred Fish Disk #344 under RKMCompanion.
