XEROX

Xerox System Integration
Guide

Introduction to Interpress

Robert F. Sproull
Brian K. Reid

XSIG 038306
June 1983

Xerox Corporation
El Segundo, California 90245

Notice

This *Xerox System Integration Guide* is an introduction to the concepts and facilities of the Interpress Electronic Printing Standard, which defines the digital representation of material that is to be transmitted to and printed on an electronic printer.

1. The contents of this document are not to be disclosed or transferred to third parties without the written approval of Xerox Corporation.
2. This guide includes subject matter relating to patent(s) of Xerox Corporation. No license under such patent(s) is granted by implication, estoppel, or otherwise, as a result of publication of this specification.
3. This guide is furnished for informational purposes only. Xerox does not warrant or represent that the Interpress Electronic Printing Standard or any products made in conformance with it will work in the intended manner or be compatible with other products in a network system. Xerox does not assume any responsibility or liability for any errors or inaccuracies that this document may contain, nor have any liabilities or obligations for any damages, including but not limited to special, indirect, or consequential damages, arising out of or in connection with the use of this document in any way.
4. No representations or warranties are made that the Interpress Electronic Printing Standard, or anything made in accordance with it, is or will be free of any proprietary rights of third parties.

Preface

This publication is one of a family of documents that collectively describe the standards underlying Xerox printing systems.

The Interpress standard defines the digital representation of printed material for exchange between a creator and a printer. A document represented in Interpress can be transmitted to a raster printer or other display device for printing, it can be transmitted across a communication network as a means of exchanging graphic information, or it can be stored as an archival master copy of the material. A document in Interpress is not limited to any particular printing device; it can be printed on any sufficiently powerful printer that is equipped with Interpress print software.

This publication provides an introduction to the Interpress standard and gives a number of examples. The first chapter is a survey of the concepts and facilities of Interpress. The remainder of the publication is directed to the system designer and programmer who are designing software to create Interpress masters. This publication is intended to be read in conjunction with the reference document for the standard, *Interpress Electronic Printing Standard*, XSIS 048306.

Comments and suggestions on this document and its use are encouraged. Please address communications to:

Xerox Corporation
Printing Systems Divison
Printing Systems Administration Office
701 South Aviation Blvd.
El Segundo, California 90245

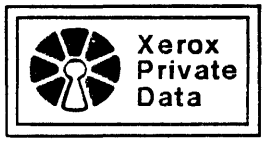
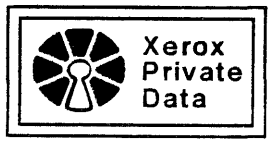


Table of contents

Introduction

Simple masters



1

Introduction

The Interpress Electronic Printing Standard is a standard for connecting raster printers to digital computers. A raster printer is an electronic device that prints on paper by making a fine systematic scan of the print area, in a manner very similar to the way a television makes a systematic scan of its picture tube. Many raster printers use a laser to perform the actual printing, and for this reason they are sometimes called laser printers.

Computer programs generate representations of documents in Interpress that are sent to printers, where the representation is processed to produce printed output. Interpress documents can also be stored in files for later demand printing, transmitted to other sites as a means of communicating complete documents, or printed on different printers at the same site when different printing qualities or speeds are required.

The Interpress standard can be used to accomplish a wide range of digitally-controlled printing:

- Computer listings. Interpress can be used to represent high-volume program listings, tabular output, and other data traditionally printed on a line printer.
- Word processing. Interpress can be used to represent the hardcopy output from word-processing equipment in the form of letters, reports, and other modest-sized documents. These documents require a certain amount of formatting versatility, multiple fonts, and so forth.
- Graphic output. Interpress can be used to describe the graphical images normally printed on plotters, such as the output from computer-aided design programs or illustrator programs. The design for a printed circuit board or an integrated circuit mask can be represented in Interpress for transmittal to a machine capable of actually making the board or the chip.
- Publishing. Interpress can be used as a means of assembling together all of the text and graphics needed for publication-quality text. This document was assembled using such a technique. An Interpress master copy of a book can be used to generate plates for a printing press or to produce single copies as needed.
- Presentations. Interpress can be used to represent the images to be projected on a screen during an oral presentation.

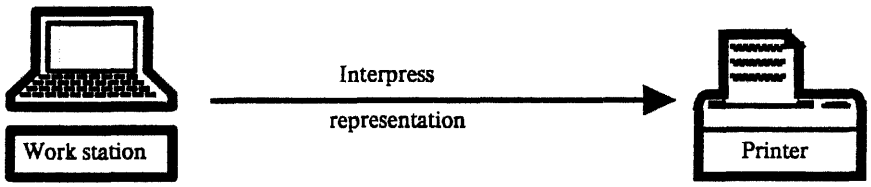


Figure 1.1. Interpress representation used for printing

Figure 1.2. Interpress representation used for archiving

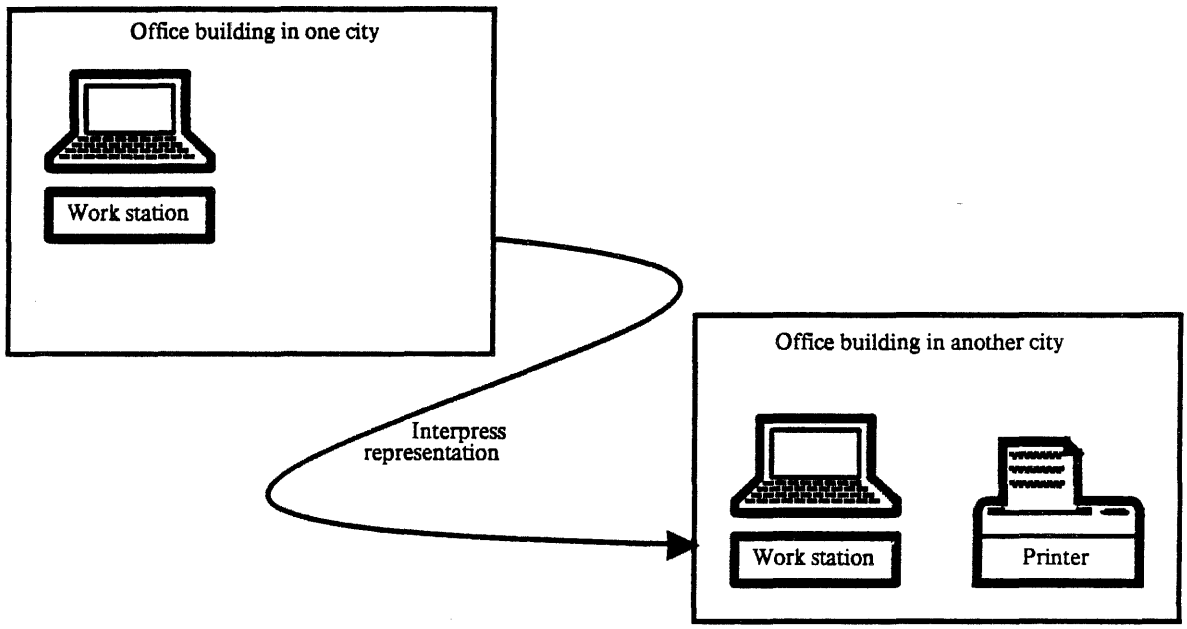


Figure 1.3. Interpress representation used for remote printing

Although Interpress is designed principally for print graphics and computer-aided document preparation, it is equally useful for specifying such diverse images as those for 35-millimeter slides, integrated-circuit masks, television stills, or frames in an interactive graphic help system. Almost any two-dimensional image can be specified by Interpress.

1.1 Why an interchange standard?

Interpress is an *interchange standard*, a means by which information in a standard format may be exchanged between a wide variety of computers and printers. Figures 1.1 through 1.3 indicate some of the intended applications of the Interpress standard. In the simplest case, the

transmittal of a document between the workstation that generates it and the printer that prints it is achieved by representing the document in Interpress. Documents so represented can be stored away in files for future reference or printing; they can also be transmitted to other sites as a means of communicating the document to a remote reader.

By standardizing the interface between the creator of a document and the printer of a document, Interpress avoids a proliferation of special-purpose software. In the absence of an interchange standard, every workstation would be required to contain software for driving a wide range of printers—any printer that a customer wishes to attach to the workstation, either directly or indirectly through a communications network. By standardizing the representation of documents to be printed, Interpress allows each workstation to have only a single interface, to Interpress. The documents generated by the workstation can still be printed on a variety of printers, but each printer is now responsible for interpreting a single Interpress interchange standard.

The definition of the Interpress interchange standard need not be concerned with the properties of the communications and storage facilities that are used to connect the document source to the printer. These facilities are said to be *transparent* to Interpress, i.e., they simply transmit or store the data without examining it or modifying it in any way. Any transparent communications or file system will do. As a consequence, Interpress specifies only format of the interchange data and not the hardware interfaces used to transmit or store it.

1.2 The nature of the Interpress standard

All computer printers are linked to the computers that control them with some sort of interface, which exchanges more information than just the data to be printed. For example, at the beginning of each line, a line printer honors carriage-control codes, which specify vertical spacing. In the past, these interfaces corresponded closely to the capabilities of the printer: if a printer could print characters of different sizes, then the interface offered a command to change size; if a printer could print superscripts, then it offered a way of specifying superscripts. The data flowing on the computer-to-printer interface consisted of device controls, including a command to control every option of the printing device.

Computer-driven raster printers are capable of printing any imaginable combination of text, graphics, and pictures—in fact, anything that can be printed with a traditional printing press can be printed with a good enough raster printer. It is therefore no longer reasonable to design an interface as a set of commands that correspond to the capabilities of some particular printer. A raster printer can print anything at all simply by arranging an appropriate pattern of black and white dots on the image, and it is the expressive power of the interface rather than the capabilities of the printer that will limit the range of images that can be printed. Since the nature of the interface is not dictated by the properties of the printing machine, it is now possible to have a universal interface, an interchange standard, in which any document at all can be represented, and that can control any raster printer. Interpress is such a standard.

A wide variety of designs could be used for interchange standards, since the properties of the printing machine no longer impose on the interface format. The following sections explain the way Interpress approaches the interchange problem.

1.2.1 Interpress is not pictorial

One possible way to define a digital printing standard might be to mimic the interface to a traditional printing press by presenting the printer with a facsimile picture of what its output is to be. The computer program would prepare data directly in raster format and the raster printer would print it verbatim. There are several disadvantages to this scheme, however:

- Raster data takes up an enormous amount of storage space—often hundreds of millions of bits to represent the image on a single page. A raster representation of a four-color halftone page from a high-quality retail catalog typically takes 160 million bits to store. This kind of storage inefficiency slows down transmission and increases storage costs. An Interpress document should be as compact as possible.
- Raster data is dependent on the resolution of the printing device—the number of dots to the inch. If the resolution of the raster printer is not the same as the resolution intended by the programmer, then the printed image will have the wrong size or shape. An Interpress document should print equally well on a wide range of printers, regardless of their resolution or scanning convention.

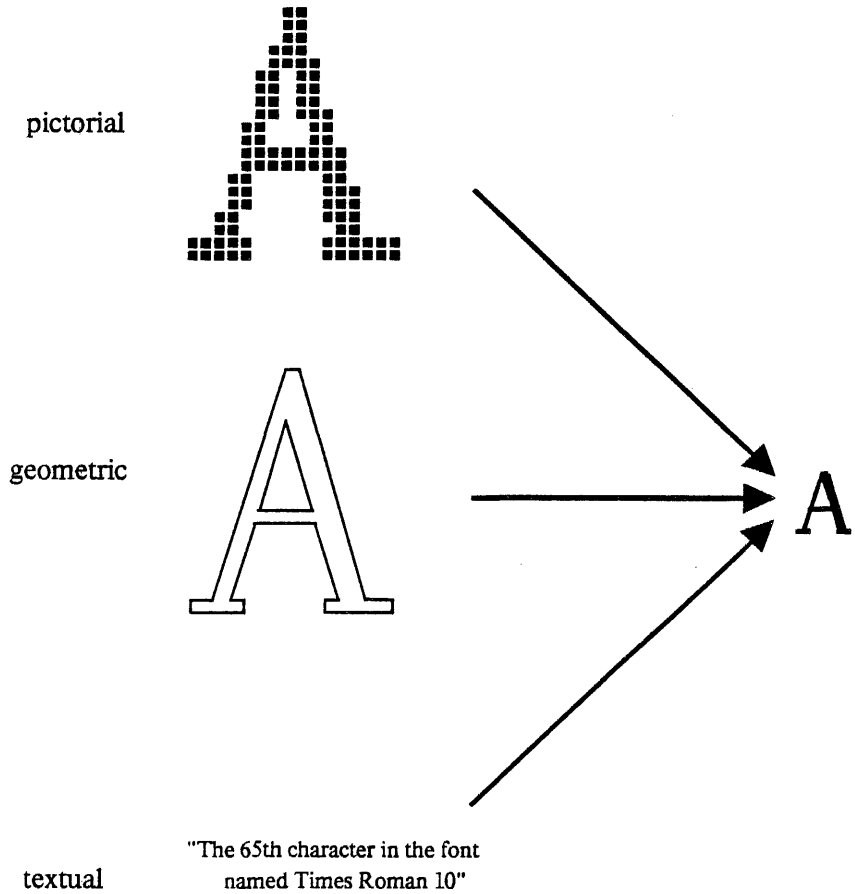


Figure 1.4. Three ways of printing the letter "A" on a page

- It is not possible to do a good job of transforming raster images—rotating them, shrinking them to fit into a particular space or to accommodate a different printer resolution, and so forth. Yet these transformations are vital to an artist or editor assembling high-quality published material.
- If the printed page is to contain text, the computer program that generates it must have access to all of the raster pictures of all of the letters in all of the fonts that it will be using. This adds a huge burden in complexity and storage to the generating program. The task of creating an Interpress document should be as simple as possible.

To help compare the approaches, consider Figure 1.4, which illustrates several different ways that a print representation could produce the same pattern on a page. The first row depicts a facsimile representation, which takes about 1500 bits to store at typical raster printer resolution. The second row shows a geometric representation, which takes about 500 bits to store the vectors in the outline. The third row shows a textual representation that instructs the printer to extract a character from a particular font; this method requires about 10 bits to represent a character. The third approach is the one preferred in Interpress.

For all of these reasons and more, Interpress is not a pictorial standard—it does not specify an image of the printed page—but is instead a set of instructions to the print software to generate that image.

1.2.2 Interpress is not static

The appearance of a paper original is limited only by the skill of the artist, who is free to use any combination of colors, shapes, sizes, typefaces, and textures that he can draw on the paper or glue to it. The appearance of the copies made from that original will be determined by the quality of the printing press. Printing processes that range from inexpensive black-ink offset printing to seven-color sheet-fed gravure printing can be used to reproduce the same original, and the quality of the resulting copy will vary accordingly.

The appearance of material printed by a raster printer is limited by the skill of the artist and by the quality of the printer, in precisely the same way that material from a printing press is limited by those factors. But since there is no paper original that the artist can use to draw and paste and color in an arbitrary way, the appearance is also limited by the ability of the artist to communicate to the computer just what it is that he wants the printer to print.

A standard for print graphics must not have built-in limitations on its expressive power. There must be no pictorial or textual combination that cannot be specified with it. Otherwise, it will eventually become obsolete. To help avoid limiting the power of Interpress, a master is represented as a program, written in the Interpress programming language, that is executed by the printing machine to produce the finished document. Although most Interpress masters will consist entirely of simple imperative statements, such as "place the letter *b* here," the full power of a programming language is available for complex applications. Programming may be used if the master must adapt to various properties of a printing device, such as whether it can print in color or only black-and-white. Programming is also used by utility programs to change Interpress masters in complex ways.

1.2.3 Interpress is not device-specific

Interpress permits documents to be represented in a way that is independent of any particular printing device. That means that as long as the printer is sufficiently powerful, the Interpress representation of a document does not depend on the details of the printer. Whether the printer is color or black-and-white, high or low resolution, continuous-tone or black-only, an attempt to print an Interpress document on that printer should yield a usable result. The goal device independence is responsible for the complexity of the Interpress standard.

Although Interpress is designed for raster printers and other highly capable devices, it may also be used with other less capable printing devices, including impact printers. When dealing with less capable "low-end" devices, there are two entirely different interpretations of the notion of device independence. The first kind of device independence views the digital representation of the document as an ideal: a specification of what the document would look like if it were printed on a perfect printer. A low-end printer must do the best it can to imitate the ideal. The second kind of device independence views the digital representation of the document more loosely—as a collection of material that is to be formatted and presented to the reader according to the best abilities of the device. In this second model, the printer's job is to make the best presentation of the material, regardless of how much that deviates from the way it would look on an ideal printer.

Each kind of device independence is appropriate in different circumstances. When the low-quality printing device is being used as a proof printer for a high-quality printing device, then the first kind of device independence is called for. When the low-quality printing device is being used as a finished-copy printer in its own right—when the material printed on the low-quality printer is the material that will actually go to press—then the second kind of device independence is called for.

Interpress addresses both kinds of device independence. It provides the first kind of device independence by treating the encoded document as the specification of an ideal appearance, which the printer then mimics as closely as possible. An important consideration in the design of Interpress is the choice of image-generation functions that allow printers to exert their best efforts to approximate the ideal appearance, and to cope gracefully with masters that contain image-generation functions beyond the capability of the printer. Certain low-end printing devices are not able to print any Interpress masters satisfactorily; these are devices that severely restrict the allowed positions of characters on the page. Most word-processing printers can print satisfactorily a document specified in Interpress that contains only text in a single font. Some printing devices, such as optical photocomposers, can do a good job of printing Interpress masters that call only for text characters, but they must leave blank space where graphics and facsimile images would appear. Finally, some equipment can print arbitrary Interpress masters with very high quality.

Interpress can also be used to obtain the second kind of device independence. In this case, the software that generates the document obtains detailed information about the capabilities of the printer that will print it, and formats the material so that it will appear pleasing on that particular device. In this case, it is not the device-independence features of Interpress that are important, but rather its function as a universal interchange format for controlling printers.

1.2.4 Interpress is not a text formatter

An Interpress printer makes no formatting decisions. The information conveyed to a printer by Interpress instructs the printer exactly where on the page to locate each character, line, image, or other graphical object. An Interpress printer is not intended to decide how long a line of text will be, to break long lines into multiple lines if necessary, or to decide how many lines will fit on a page. These formatting calculations, if they are necessary at all, are performed by the software that converts a document into Interpress format rather than by the printer. During this conversion process, which is sometimes called *composition*, the software decides where on the page every object is to lie and then uses Interpress to convey this information to the printer.

1.3 Overview of the Interpress standard

The representation of a document in the Interpress standard is called a *master*, a term chosen by analogy with conventional printing reproduction techniques. Any computer program that generates an Interpress master is called a *creator;* any program that interprets a master to make an image is called a *printer.*

The master can be viewed as a set of commands to instruct the printer how to construct the image of each page of the document. The function of the printer is much like that of a draftsman: it is presented with detailed instructions for constructing an image, not with the image itself.

The master is actually a program coded in the Interpress programming language and represented digitally as a sequence of bytes. The procedural nature of the master is entirely invisible to someone who is using an application program that generates Interpress output—a user is free to think of an Interpress master as a data file that, when sent to an Interpress printer, will produce printed output. However, a programmer who needs to write a creator program must be aware of Interpress' procedural nature. Each Interpress printer is an interpreter for the Interpress programming language; it interprets the operands and operators in an Interpress master to produce actual printed output. The printer is said to *interpret* the master or, equivalently, to *execute* it.

The Interpress language definition is divided into two parts. One part is the description of the *imaging operators* in Interpress—the operators that build up an image by placing characters, drawing lines, inserting halftones or facsimile images, and so forth. The other part is the description of the *base language*, which is a set of rules for recording invocations of imaging operators in a master. Although the imaging operators are the heart of Interpress, the base language is described first so that it can be used in the descriptions and examples of the imaging operators.

1.3.1 The base language

Like all programming languages, Interpress has both syntax and semantics. The semantics of the Interpress language are the rules for how the various operators behave when they are executed by the printer, and the syntax of the language is the set of rules for how the calls to those operators are coded in a master. Because Interpress masters are intended to be created and interpreted by programs and not by people, the syntax need not be elegant or even very

readable, and in fact the syntax of an Interpress master is designed to make it easy for programs to produce and easy for programs to interpret.

Interpress is a stack-oriented language, and we can most conveniently view its operators in a postfix notation. Postfix is a notation like that used on some hand-held calculators, in which the operands are pushed onto a stack and then the operator is executed, whereupon it removes its operands from the stack and replaces them with its result(s). The major syntactic difference between an infix notation and a postfix notation is that postfix operators follow their arguments while infix operators usually appear between their arguments and use parentheses to indicate the order of operations. For example, the computation represented by the infix notation 1+1 might appear in postfix as

```
1 1 ADD
```

Changing the name of the operator "+" to ADD is unimportant; what is important is reordering the notation so that the operator appears last; hence "post" fix. By way of a second example, the computation (4*14)+6 might be represented in postfix as

```
4 14 MUL 6 ADD
```

To understand this second example, we must know that the multiplication operator MUL requires two operands (the 4 and the 14), and returns one operand (their product), and that the addition operator ADD also requires two operands (the product and the constant 6).

Some operators return no results at all on the stack. These operators are usually executed because they have an important *side effect*, an effect not on the stack but on some other piece of the interpreter. For example, the program

```
-12843 12 FSET
```

sets the value of the 12th frame variable to −12843, but leaves no result on the stack (frame variables are explained later). By far the most interesting operators that are executed for side effect are the imaging operators that are used to build up an image to be printed.

1.3.2 Imaging operators

An Interpress printer interprets, or executes, a master in order to print the document it represents. During that execution, the printed document is built up one page at a time, and each page is printed before the next one is begun. The printer maintains a *page image* inside it, which is a pictorial representation of what the page would look like if it were printed at that moment. The *imaging operators* make changes to the page image.

A complex image is built up by starting with a blank page image and making a sequence of changes to it. Each individual change is a simple one, usually adding a single graphical object to the image. Images can be built up by placing on the page text characters obtained from font libraries, possibly scaling or rotating them before placement. Images can be built up by drawing lines, which can range from horizontal and vertical lines to complex polygons. Closed areas defined by polygons can be filled with ink. Facsimile pictures, including halftones, can be placed on the page. Images are built up using arbitrary combinations of these graphical primitives.

The task of maintaining and altering the page image is undertaken by the *imager*. Thus the functions of the printer divide neatly into two parts: a base language interpreter, and the imager. As the master is executed by the interpreter, calls are made to imaging operators in the imager, which result in building up the page image. Operands for imaging operators are passed in the stack, just like operands to arithmetic operators. In addition, imaging operators refer to information held in some *imager variables*. These variables are conceptually like additional arguments to imager operators, but are kept separate to reduce the number of arguments to frequently-used operators. Information that changes only occasionally is kept in imager variables, while information that is likely to be different for each call on an imaging operator is provided as a stack operand to the operator.

An important imager variable is a *current transformation*, denoted by the symbol *T*. The current transformation is a linear transformation, a sort of mathematical lens, that is applied to all operators that specify a position on the page. If the current transformation is set to something that magnifies by a factor of 2, then all distances and character sizes will be automatically doubled. If the current transformation is set to something that leaves the size alone but rotates 90 degrees to the right, then all images generated will be rotated 90 degrees to the right. The versatility of Interpress is due in large part to the use of transformations—they provide a uniform way to achieve a wide range of effects, such as obtaining characters of different sizes and orientations, reducing or rotating an entire page image, combining separately prepared illustrations and text into a single master, and so on.

One of the most commonly used imaging operators is the one that "shows" a string of text on the page. This operator depends on some setup for selecting a typeface, a transformation that will determine the text size and rotation, and a starting position for the text string. The following program fragment will print on the page the string "Interpress":

```
0.07366 0.23876 SETXY
<Interpress> SHOW
```

The first line shows one of the setup operations, namely setting a starting position for the text string. The second line places on the stack a representation of the character string "Interpress" and calls the imaging operator SHOW to place the characters on the page image.

1.3.3 Encoding

An Interpress master is *encoded* as a sequence of digital values (bytes) that can be transmitted to a printer. Most computer programs are represented as a sequence of text characters, which are then encoded into digital values using a character set, a correspondence between each character and a digital value. For a number of reasons, this approach is not used in Interpress. One important reason is that text representations are not sufficiently compact when a great deal of numeric data must be encoded, as for example in a facsimile image. A second reason is that the process of decoding the program from the text (usually called lexical analysis) is cumbersome and slow. And finally, since Interpress programs need to be read only by computer programs and not by people, the text form offers little advantage.

The Interpress encoding specifies rules for representing each *literal* of an Interpress master by one or more 8-bit bytes. A literal is an instance of a particular type of object that occurs in the program. For example, ADD and MUL are *primitive operator literals;* "147" and "−12483" are *number literals*. Encoding rules specify how each literal will be represented. For example, the

FSET literal is represented by a single 8-bit byte whose decimal value is 149. The MUL literal, which is used less often, is encoded as a two-byte sequence: 160, then 210. The encoding rules are designed so that the Interpress interpreter in a printer can quickly decipher the sequence of bytes in the master and recover the literals.

Examples of Interpress masters, such as those that appear throughout this document, are presented using a textual notation, more in keeping with conventional programming languages. While examples might be more accurately presented as a sequence of encoded 8-bit values, they would be very difficult for people to read. The difference between our textual notation and the Interpress encoding is somewhat akin to the difference between an textual assembly language and its corresponding binary encoding of computer instructions and data. More details on the presentation of examples will be introduced as they are needed.

1.4 How Interpress masters are printed

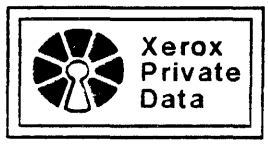
Each printer that prints Interpress masters is an interpreter that executes the Interpress language contained in a master. That execution produces the desired printed output. Most, but not all, of the information required to print a master is provided in the master itself. Two kinds of information supplement that which is provided in the master: data from an *environment* maintained by the printer and *printing instructions* that specify details of the printing request.

1.4.1 The environment

The environment is a library of data that a printer may refer to when printing a document. The principal use of the environment is for character fonts: most masters will request characters to be placed on the page image, but will not wish to specify in detail the exact shape of a character. In these cases, the master specifies the size, position, and rotation of each character, but relies on a font defined in the environment to specify the shape.

Each Interpress printer may have an environment that differs from environments on other printers. There are a number of ways in which environments may differ. First, the collection of fonts contained in a printer's environment may differ from that of another printer. Second, it is possible that two printers both have a font named "Times Roman," but the fonts differ significantly. This situation already arises frequently in phototypesetters, where one manufacturer's "Times Roman" differs from another's. And finally, even if fonts are carefully named so that each design has a unique name, printers may have different *versions* of a font—one may incorporate slight adjustments in order to improve its appearance on the printing device.

The environment obviously raises some problems for device independence. If the environments on two printers differ, the same master may produce different images on the two devices. This problem will be muted in practice by conventions that will tend to make environments similar or identical. For example, printers from the same product line and installed in the same organization will usually have identical environments; the printer manufacturer will provide some fonts as a standard part of the product, and the organization may have an additional font containing special symbols and logotypes installed on all its printers. In this way, masters created and printed within the organization will print correctly. If a master is transmitted to an organization that has configured its environments differently, it may be printed differently. Unfortunately, it is simply not practical to standardize environments so as to guarantee absolute device independence.



1.4.2 Printing instructions

When a master is presented to a printer for printing, it is usually accompanied by some *printing instructions.* Examples of instructions are the number of copies to be made, the name of the person who requested that the document be printed and to whom it is to be delivered, the kind of finishing required (e.g., stapling, binding, collating), the kind of paper or stock on which to print, the identity of an account to be charged, and perhaps special security measures to prevent others from seeing the document. Interpress defines a mechanism for specifying printing instructions in the master as well as in a printing request.

1.4.3 Subsets

Because the Interpress standard is so general, there will certainly be cases where it is necessary for a printer to ignore some parts of a master. For example, a raster printer with very simple software might honor only horizontal and vertical lines while ignoring lines at any other angle. To provide a certain amount of regularity for printers that must implement only subsets of Interpress, the notion of *subset* is defined. All Interpress printers must support the *text Interpress* subset, which includes text, horizontal and vertical lines, and black ink. Other subsets support arbitrary graphics, scanned images, shades of gray, and colors.

1.5 Manipulating Interpress masters

Paper masters can be cut and pasted, photographed and scaled, and manipulated in other ways. The programmability built into Interpress allows a master to be created that will automatically adapt itself to varying circumstances by testing its environment and creating different images accordingly. Nevertheless, there are many sorts of manipulations that either cannot be anticipated at the time the master is created or are too complicated to build into the master at creation time. These operations are performed by means of what are called *master manipulations*, and are carried out by one or more *utility programs.*

The Interpress language is carefully designed to make master manipulations tractable without requiring the utility program to understand the details of the master that it is manipulating, i.e., the manipulation is purely syntactic. The simplest utility program need only be able to locate the major components of a master and copy them bodily. More complicated manipulation may involve rearrangement of pieces of various masters, possibly separating one master into several or merging several masters into one. Some manipulations may require the utility program to look inside the components of a master and change what it finds there.

Here are three examples of Interpress master manipulations, listed in increasing order of their complexity. The first example requires only physical replication of parts of an existing Interpress master; the second example requires intelligent substitution of one master into an appropriate spot in another. The third example requires that a geometric transformation inside the master be modified and that pairs of adjacent pages be combined.

As our first example, let us suppose that we have an Interpress master that represents an 11-page document. A new document is to be created, containing pages 1, 2, and 4 from the original. As shown in Figure 1.5, the original master is passed through a utility program that creates a new master by copying the parts of the original master that define pages 1, 2, and 4.

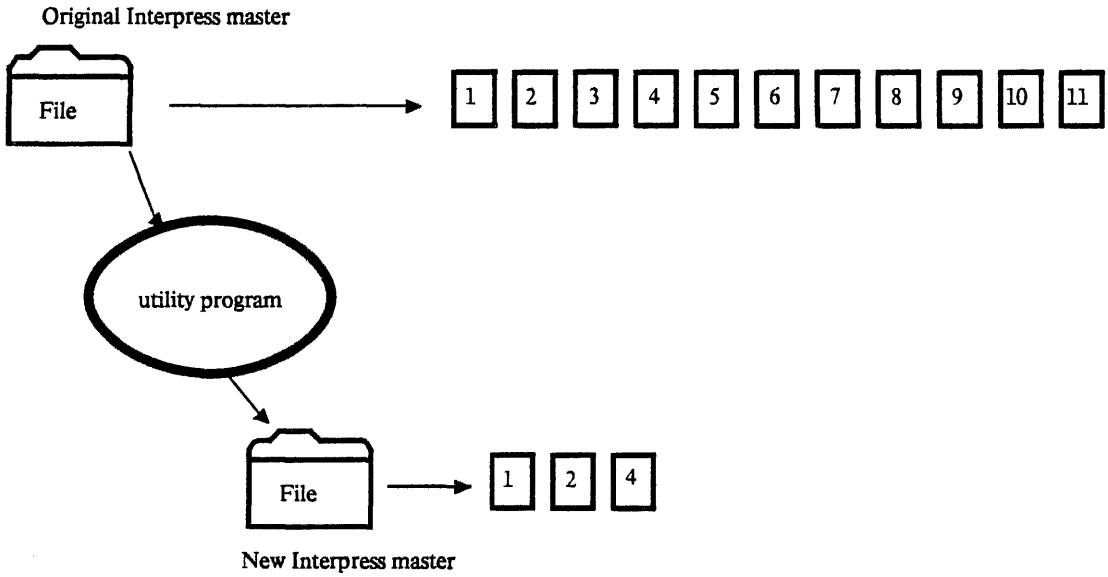


Figure 1.5. Selecting pages from an Interpress master

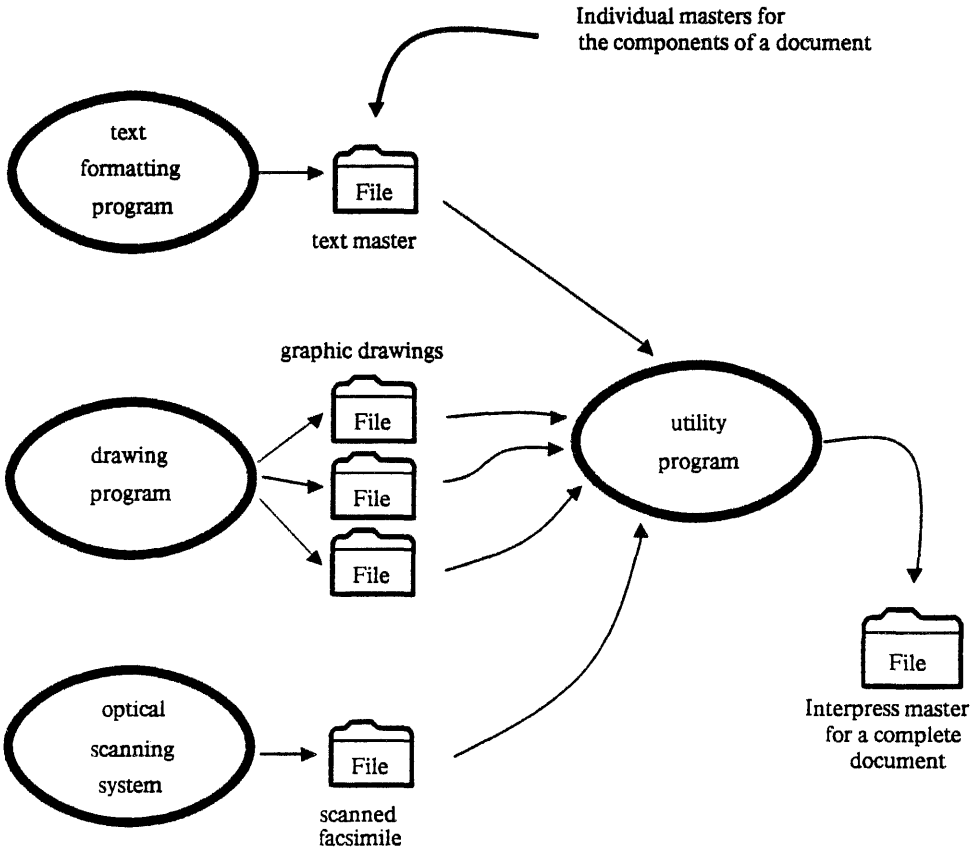


Figure 1.6. Merging separately-created pieces into one master

As a second example, suppose that a document like this Interpress introduction you are reading is being produced, containing both text and figures. The authors of this hypothetical document choose to use separate programs to create the pictures and the text, and in fact one of the pictures is a digitized photograph. Figure 1.6 shows the manipulation that is used to merge the complete set of files together into a single document.

As a third and final example of manipulating Interpress masters, suppose that you have a master for a document intended to be printed on 8½ × 11 paper, such as this Introduction. Suppose further that you need to print that same document in what is sometimes called a "two-up signature" format, in which two pages' worth of text are printed on each side of each piece of paper, shrunk down to about half of their former size and rotated 90 degrees to the left. This manipulation can be done by changing the geometric transformation associated with each page in the Interpress master to cause a rotation by 90 degrees and a size reduction by a factor of 0.65 to be applied to the page. The various page definitions are then rearranged and combined so that when the sheets are printed, folded up the middle, and stapled, the pages will come out in the right sequence. This operation is shown in Figure 1.7. Note that an edge of each signature page is unused because of the paper shape and that the order of the pages has changed to compensate for the signature binding. More complex master manipulations of this sort can be used to assemble images for 16-page signatures such as those commonly used in the printing industry.

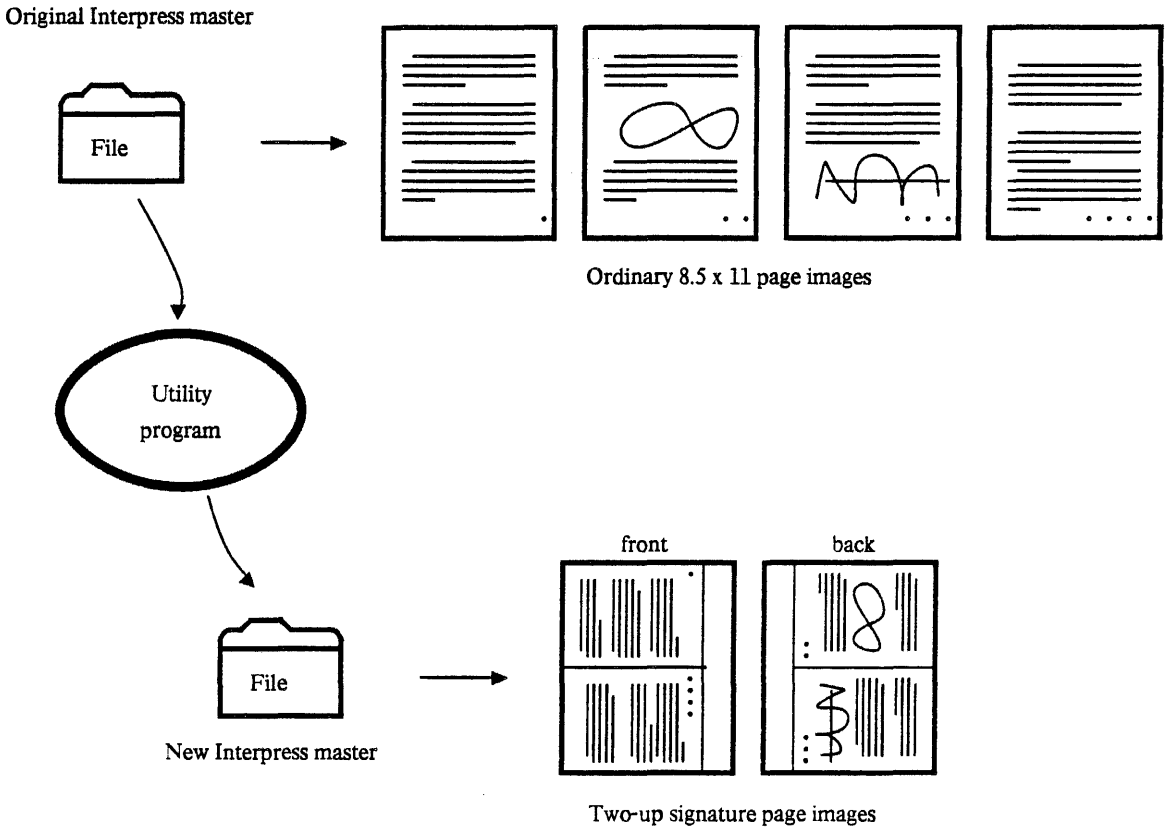


Figure 1.7. Preparing two-up signature pages

Some manipulations of Interpress masters are so frequent and simple that they can be performed "on the fly" by certain printers. For example, a high-speed printer can print selected pages from a master, making a selection of pages according to printing instructions supplied in the printing request.

Manipulating a master with a utility program is not the same as editing the document with a document-preparation system. Master manipulations are not intended for fixing typographical errors or inserting new text, because these changes will require formatting decisions, and could lead to repaginating the entire document. That is the job of a text formatter, not of an Interpress utility.

1.6 Summary table

Interpress Goals

- Standard representation of information to be imaged on a raster printer
- Independent of printer properties
  - Resolution
  - Coordinate system orientation and units of measurement
  - Special capabilities such as color, scanned images, graphics
- Applications
  - Computer printing
  - Word processing
  - Graphical and image output
  - Publishing
  - Engineering drawings
  - Video displays . . . and more
- Non-pictorial representation
  - Reduces storage requirements
  - Avoids resolution dependence
  - Provides transformation capability
- Represented as statements in a language
- Mechanisms to control printing, accounting, and distribution functions
  - Number of copies to print
  - Copies may differ slightly, e.g., for addressees
  - Print one side, both sides
  - Finishing information, e.g., stapling
  - Break page shows document name, requestor, etc.

Interpress Non-Goals

- Not a composition system
  - Documents must be pre-composed
  - Will not change line-ending, page-ending, or other formatting decisions
- Not a document representation
  - Not intended for general-purpose editing
  - Used as an "output" format only
- Not intended to be created by humans
  - Intended to be computer-generated as output of an editing or composition system

1.7 Guide to Interpress documentation

Several forms of documentation about Interpress are available. Each serves a different purpose.

The *Interpress Electronic Printing Standard* [26], hereafter called "the Standard," is the reference document for the current version of the Interpress standard. It is written with precision and accuracy in mind and as a result is not easy to read. The Standard is the authoritative definition of Interpress; other documents are secondary. If errors in the definition of Interpress are discovered, they will be repaired by new editions of the Standard.

The *Interpress Reader's Guide* [27] is a companion to the reference document. It provides commentary on various aspects of that document.

The *Introduction to Interpress*, the document you are reading, is a tutorial that presents all but the most advanced concepts of Interpress. It is designed to be read sequentially and has numerous examples of correct Interpress masters. When this *Introduction to Interpress* refers to itself, it will be called "the Introduction."

1.7.1 Guide to this document

A basic understanding of Interpress can be obtained by reading Sections 2–8 of this document. Sections 9–17 treat more complex aspects of Interpress. Section 17 is a collection of hints and suggestions for constructing Interpress masters. Sections 18–21 deal with printing Interpress masters. Section 22 describes how Interpress can be used as part of larger document-handling systems. Finally, Section 23 presents some of the rationale behind the design of Interpress.

A reader who desires only a general understanding of Interpress is encouraged to read Sections 1–8 and as much of the remainder of this document as possible. Section 22 is also recommended.

A reader who intends to write a program to create Interpress masters is counseled to read this entire *Introduction to Interpress* along with the Standard. The Introduction is not self-contained; frequent reference to the Standard will be required.

The Introduction contains cross references both to itself and to the Standard. References to other sections of the Introduction are mentioned using the spelled-out word "Section," for example "see Section 3.1." References to the Standard are mentioned with the section sign "§," as for example "see § 2.5." The Standard contains a very helpful glossary, which is not reprinted in this document.

Fine-print passages such as this one appear occasionally. They usually contain detailed discussion of topics that are not central to understanding Interpress. It is probably best to ignore them on a first reading.

Throughout this Introduction, Interpress is described in a simplified, often incomplete way. This style is essential to an introduction, where complexity must be avoided to ease comprehension. The reader is warned that this strategy will oversimplify Interpress. The Standard is the only complete documentation for Interpress.

2

Base language I

The Interpress standard consists primarily of a set of imaging operators embedded in a programming language called the Interpress base language. The imaging operators can be regarded as built-in functions and the base language as the computational framework that invokes them.

The base language is a programming language in which programs will not be written by people, but will be created by other programs. In that regard, it is more like an instruction set than a programming language. Furthermore, programs in the base language will be executed by the support computers attached to printers, which may not always be fast or powerful. The base language is therefore somewhat primitive by the standards of modern programming languages.

The base language is thus best understood as a machine language for a particular stack-oriented postfix machine that supports a number of imaging operations in addition to its run-of-the-mill instruction set. In the absence of a computer custom-tailored to this machine language, the Interpress machine is simulated by an interpreter running on an ordinary computer. We say that this interpreter is an implementation of the Interpress virtual machine, which is to say that the interpreter can be used to execute Interpress masters by simulation.

The Interpress base language, as a machine language for the Interpress virtual machine, shares with other machine languages three properties that make it different from ordinary programming languages:

- The language has minimal syntax.
- An Interpress program does not use identifiers for naming variables, procedures, and the like. (The language does support identifiers, but they are used for referencing objects outside the Interpress program, in the environment.)
- An Interpress program need not be compiled or pre-scanned in order to be executed; it can be interpretively executed one literal at a time.

The Interpress base language is conceptually similar to the *Forth* language that is popular among microcomputer users. Its execution is very straightforward: when the interpreter encounters an operand, it pushes it onto the runtime stack. When the interpreter encounters an operator, it executes it. Most operators pop some number of operands from the stack, compute with them in some way, and then push some number of results back onto the stack.

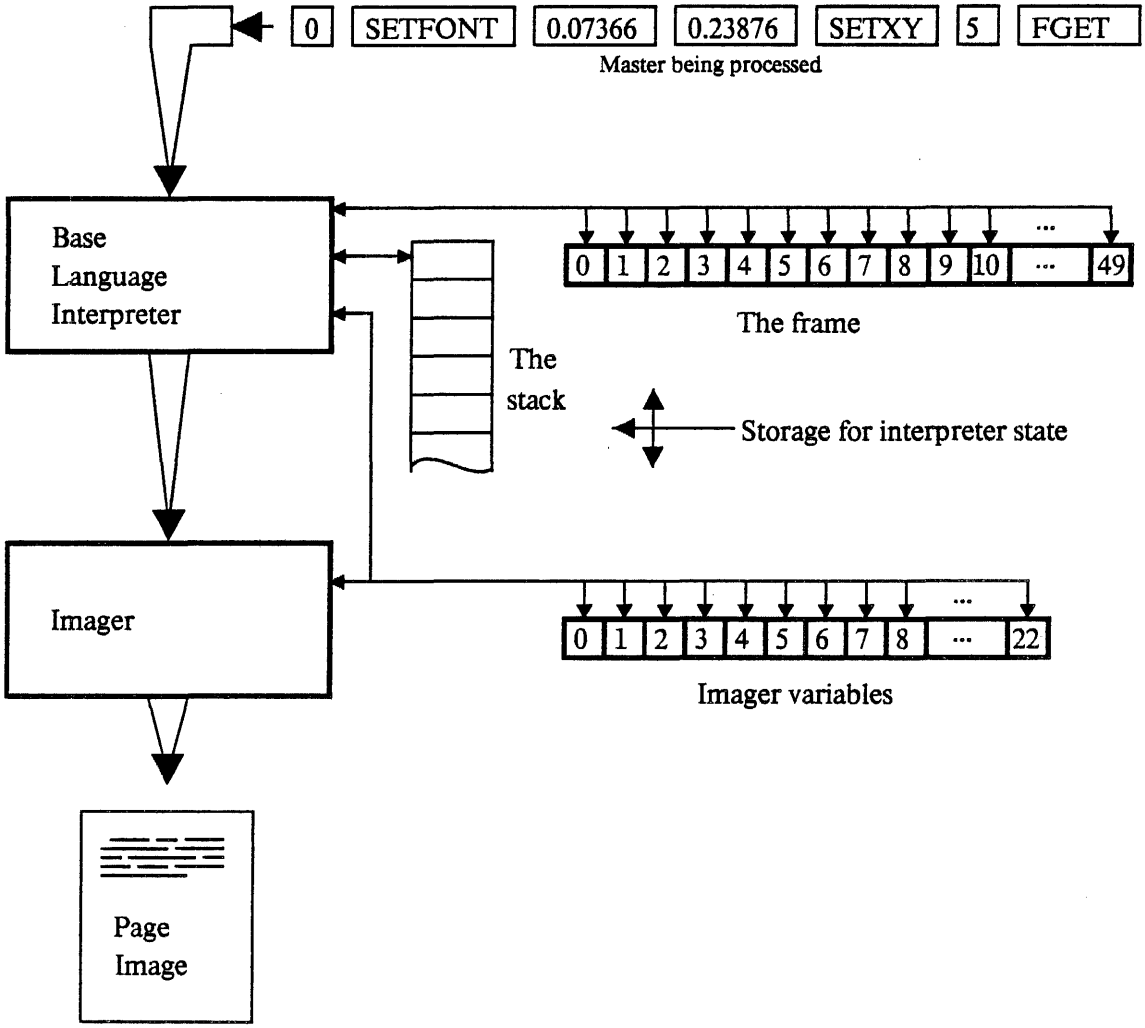


Figure 2.1. Structure of an Interpress printer

An important consideration in the design of the base language is that it must be interpreted identically on a range of printing equipment. Each Interpress printer will have a small computer attached, whose job is to interpret Interpress masters. Since different printing equipment may use computers with different properties, it is important that the Interpress base language specify interpretation rules in a way that does not depend on the computer's properties. These requirements are the source of some of the complexity in the Standard.

This section introduces a few of the constructs of the base language. More of the base language is covered in Section 12. The base language is described in detail in § 2. Figure 2.1 shows a rough diagram of the major components of the base language's computation environment.

2.1 Types in the Interpress base language

While the Interpress base language is similar to that of a pocket calculator in that a stack and operator postfix interpretation scheme is used, Interpress manipulates values of several different types, while a calculator manipulates only numbers. Interpress allows about a dozen different types of values to appear on the stack and to participate in interpretation. For the moment, only a few will concern us.

The first three types are similar to types found in conventional programming languages:

- Number. A number is a computer representation of a rational number; it is like a *floating-point number* in conventional programming languages. While the discrete and finite nature of digital computers does not allow all rational numbers to be represented (§ 2.2.1, § 5.2), Interpress guarantees that a good deal of precision is maintained.

  An Integer is a non-negative integral Number. Integers do not constitute a distinct type, but rather are a subset of Numbers.

  The Standard makes a distinction between Integer (capitalized) and integer (not capitalized). The capitalized form is given a formal definition in Interpress: Integers are greater than or equal to zero. The low-case integer, on the other hand, is simply an integral Number, which may be negative. This distinction is historical, and somewhat unfortunate.

- Vector. A vector is a sequence of values; each value in the sequence is called an *element* of the vector. An element in a vector is named by its *index*, an Integer. An Interpress vector is like a cross between the arrays and records that are present in a conventional language such as Pascal. Like Pascal's arrays, Interpress Vectors have elements that are named by numeric indices; like Pascal's records, Interpress Vectors may have elements of different types.

- Primitive Operator. A primitive operator is a function built into the Interpress virtual machine, just as the "+" and "*" operators are built into most programming languages. It is by executing operators that work gets done. There are two kinds of primitive operators: base-language operators and imaging operators. Among the base-language operators are conventional functions such as ADD and MUL. The imaging operators are those that are concerned with changing the page image to create an output image; for example, the imaging operator SHOW places characters on the page image.

The following types in Interpress do not appear in most programming languages:

- Identifier. An Identifier is a name, represented as a sequence of lower-case letters, digits, and the minus sign "−", which must begin with a letter. While most programming languages use identifiers, usually identifiers are not *values* that can participate in computation; by contrast, Interpress passes identifiers as operands to certain operators. Identifiers are not used very frequently in Interpress masters, but almost every master has some.

- Body. A Body is a sequence of literals. Since we haven't yet defined a literal, it's a bit difficult to envision a body. For those familiar with Algol-like languages such as Pascal, a body is like a sequence of statements contained in a BEGIN-END block. The notion of bodies should become clear in Section 3.

A great many Interpress masters can be created using only these five types. Details of the domains of these types are given in § 2.

2.2 Literals

Literals are the primitives from which a master is made: a master is a sequence of literals of various types. Number, Operator, Identifier, and Body literals may all appear in Interpress masters. The term *literal* conveys the notion that a value of that type is literally present in the master, as distinct from a computed value that may exist only on the stack during interpretation.

In order to represent literals in the master, they are *encoded* into a sequence of 8-bit bytes. It is also useful to have a *written form* of each literal, so that examples of Interpress masters can be exhibited in this and other documents without requiring the reader to decode the long sequences of numeric values that result from the encoding rules. Encodings are explained in Section 1.3.3 and § 2.5.

2.2.1 Written form

Conventions for writing a literal value are associated with each type that can have a literal in the master:

- Number. Number literals are written using the conventional decimal notation as a string of decimal digits, with an optional leading minus sign "−", and optionally containing a decimal point. Thus −2, 1734, 205781302216, 14.5, and −3.2 are all Numbers.

  In the Standard a number may also be written as a quotient of two integers, separated by the "/" character. Thus −11/3 and 10/−3 are Numbers. Moreover, the literals 31/10, 93/30 and 3.1 have the same value. This quotient form will be used rarely in this document.

  The decimal fraction notation is used only in the written form. So-called floating-point numbers do not appear in a master encoding. Instead, each decimal fraction is converted to the quotient form, which requires only integer representations. For example, 1.25 can be converted to 125/100 and 1.1 to 11/10.

- Primitive operator. A primitive operator literal is written by giving its name as a word in upper-case letters and digits, such as MUL, ADD, POP, COPY. A glance at § B.2 will reveal all the Interpress primitive operators. Beware that words that look like operators but begin with "*" represent operators that have no literal representation; these symbols are used to represent operators internal to the printer but inaccessible to the master. These are included in the Standard because they are useful in explaining the semantics of other operators.

- Identifier. An identifier literal is written as a word that has at least one lower-case letter in it; it may also contain digits and the minus sign "−". Thus *Helvetica*, *version−2*, and *Xerox* are all identifiers.

- Body. A body literal is a sequence of literals, and is written by enclosing the written form of the sequence inside brackets { }. For example, { 1 13.4 MUL } is a body literal. A body is *not* a vector; a body is like a piece of a program, while a vector is a subscriptable collection of data. A body contains only literals, while a vector contains only values. The purpose of a body is to be executed at some point during the printing of a master. It corresponds somewhat to the familiar BEGIN/END blocks in other programming languages.

The list above is actually a complete list of all the literals that can appear in an Interpress master. There are additional data types in Interpress, such as Vector, but these have no literal forms. Instead, these types are *constructed* using primitive operators, a process explained further in Section 2.4.

Comments may be interspersed among literals in the written form. Any text between pairs of double-minus-signs is a comment. Thus *--this is a comment--* is a comment. Comments are used extensively in this document to help explain the contents of a master. Often a line of an example will begin with a comment that identifies the line number, e.g., --14-- so that the accompanying text can refer to it.

When the literals of a master are written in an example, they are usually formatted so that they can be easily read; this formatting is irrelevant to the contents of the master being illustrated. New-line characters, tabs, and spaces may all appear between literals, so that Interpress masters in the written form can be prettyprinted or formatted in any style that will enhance their readability.

Several software tools, described in Section 8, produce or interpret the written form.

2.2.2 Encoded form

While most programming languages encode programs as a string of text characters, Interpress uses a more compact binary encoding. Just as there are rules for writing each literal in an example, there are rules for encoding each literal as a sequence of 8-bit bytes. It is this sequence that is a real Interpress master: it may be stored, transmitted to a printer, and so forth.

You do not need a detailed understanding of the encoding rules in order to understand the rest of Interpress. All of our discussion and examples will deal with the written form of literals, with the understanding that these same literals can be encoded using the Interpress encoding and thereby become a real master.

If you're curious about the encoding rules, glance at § 2.5 and at the example in Section 3.3, which shows an absurdly simple master in its written and encoded forms. Note that the software aids described in Section 8 include a written-to-encoded converter and an encoded-to-written converter. The written-to-encoded conversion is analogous to the assembly process in a traditional assembler language, and the encoded-to-written conversion is like the disassembly process performed by machine-code-level debuggers.

2.3 Interpretation rules

The execution of an Interpress body progresses sequentially, processing one literal at a time. The processing of each literal in the body depends on the literal's type. When a literal of type Number, Identifier, or Body is encountered, the value corresponding to the literal is placed on the stack. (Don't worry for now about body values on the stack—that rare case will be explained when it crops up in an example in Section 12.) When a primitive operator literal is encountered, it is executed. What happens depends upon the definition of the primitive operator. As you can imagine, the actions of the primitive operators are the heart of Interpress.

We're now ready to explain a real example. This isn't a complete master, but it is a legal Interpress body.

```
--Example 2.1--
--0--     {              --a bracket that begins the body literal--
--1--     73             --a Number literal, which is also an Integer--
--2--     13.5           --a Number literal--
--3--     -1.5           --a Number literal--
--4--     ADD            --a primitive operator literal--
--5--     0              --a Number literal, which is also an Integer--
--6--     1              --a Number literal, which is also an Integer--
--7--     MAKEVECLU      --a primitive operator literal--
--8--     }              --the bracket that ends the body literal--
```

To make it easy to refer to the literals in this body, each literal is placed on a separate line. Since this is the first example of the Interpress interpretation process, we shall present a step-by-step history of the interpretation. In the table below, each line corresponds to the processing of a single literal in the body. The table shows the line number, the literal processed, and the contents of the stack after the literal has been processed. The stack is represented as a list of values inside special brackets ⟨ ⟩. The value at the top of the stack is the rightmost value in the list. The stack is initially empty.

Line	Literal	Stack after processing literal
1	73	⟨73⟩
2	13.5	⟨73⟩ ⟨13.5⟩
3	−1.5	⟨73⟩ ⟨13.5⟩ ⟨−1.5⟩
4	ADD	⟨73⟩ ⟨12⟩
5	0	⟨73⟩ ⟨12⟩ ⟨0⟩
6	1	⟨73⟩ ⟨12⟩ ⟨0⟩ ⟨1⟩
7	MAKEVECLU	⟨vector of 2 elements; element 0 is 73, element 1 is 12⟩

It is clear from this example that a detailed description of each primitive operator is required in order to predict how an Interpress body will be interpreted. The Standard provides such descriptions. The description of ADD, for example, is (from § 2.4.9):

⟨*a:* Number⟩ ⟨*b:* Number⟩ ADD → ⟨*c:* Number⟩
where $c = a+b$.

The first line of the description shows how the ADD operator affects the stack. The items to the left of "ADD" show a picture of the stack before the ADD operator is executed: a Number will be on the top of the stack; it will henceforth be named *b*. Another Number will be next from the top, henceforth named *a*. The ADD operator will pop both of these elements from the stack at the beginning of its execution. To the right of the → symbol is a picture of the stack after the ADD operator finishes execution: a Number named *c* will be pushed on the stack. The remainder of the description, starting with the **where** clause, describes how the result, *c*, is determined from the arguments *a* and *b*. In Example 2.1, above, *a* will have the value 13.5, and *b* the value −1.5, so *c* will be 12. The names *a*, *b*, and *c* do not appear in a master or on the stack, but are used as part of operator descriptions solely to identify arguments and results.

To continue with Example 2.1, the MAKEVECLU operator constructs a vector from an arbitrary number of arguments taken from the stack (from § 2.4.3):

⟨x_1: Any⟩...⟨x_n: Any⟩ ⟨*l:* Integer⟩ ⟨*u:* Integer⟩ MAKEVECLU → ⟨*v:* Vector⟩
where *v* is a vector with lower bound *l* and upper bound *u*. Let $n=u-l+1$. After *u* and *l* are popped off the stack, *n* additional values are popped; call them x_n, ..., x_1, where x_n is the first value popped and x_1 is the last value popped. The elements of *v* will have the values x_1, ..., x_n; i.e., the element with index *l* is x_1, the element with index $l+1$ is x_2, and so on up to the element with index *u*, which is x_n.

The MAKEVECLU primitive operator constructs a vector, using information on the stack to specify the lower and upper bounds of the vector and to specify the value of every element in the vector. In Example 2.1, the top element of the stack is 1, a Number but also an Integer. So the value of u used in the execution of MAKEVECLU in the example will be 1. Similarly, the value of l will be 0. The resulting vector will therefore have lower bound 0 and upper bound 1 and will contain two elements. The values of the two elements x_i are taken from the stack as well. The value of the element whose index is 0 will be 73 and the value of the element whose index is 1 will be 12. Notice that the definition of MAKEVECLU allows an element to be of type Any, i.e., almost any type that can appear on the stack. This means that different elements of the vector can be of different types.

Example 2.1 is really quite clumsy and should be simplified. The ADD operator is doing unnecessary work: since both its arguments are literals, the result could just as easily be computed by the creator before the master is constructed. Also, § 2.4.3 shows that the MAKEVEC operator is more convenient than MAKEVECLU when the lower bound l is zero: rather than l and u, MAKEVEC requires only the total size of the Vector. So we obtain the equivalent body:

```
--Example 2.2: same effect as Example 2.1--
--0--      { 73 12 2 MAKEVEC }
```

2.4 Constructor operators

Several types in Interpress have no literal representation in a master. Instead, Interpress provides *constructor operators* that create a value of a certain type from values of more primitive types. Although most of the Interpress types that must be constructed have not yet been introduced, we have seen that vectors have no literals and must therefore be constructed. As illustrated in the preceding section, the operators MAKEVEC and MAKEVECLU are used to build vectors from their elements.

Vectors are constructed so frequently in Interpress that special provisions are made for invoking the constructor operators. In examples, we write [x_0, x_1, . . . x_{k-1}] as shorthand for x_0 x_1 . . . x_{k-1} k MAKEVEC, which constructs a vector of k elements. This notation can be used to restate Example 2.2, which now becomes:

```
--Example 2.3: same effect as Examples 2.1, 2.2--
--0--      { [ 73, 12 ] }
```

In another form of shorthand, we write <*sequence of characters*> to construct a vector containing the "character codes" of the sequence of characters. The notation <*sequence of k characters*> stands for n_0 n_1 . . . n_{k-1} k MAKEVEC, where n_i is an Integer whose value is the code for the ith character of the sequence of characters. The mapping from characters to codes depends on considerations that will be explained later. As an example, if we use the ISO 646 [10] character code mapping (very similar to ASCII [1]), the written form <Xerox> is equivalent to [88, 101, 114, 111, 120], or, to reduce the example to literals, 88 101 114 111 120 5 MAKEVEC.

The rules for encoding Interpress masters into a sequence of 8-bit bytes provide two compact ways to represent constructions of vectors of Integers (§ 2.5.3, *sequenceLargeVector* and *sequenceString*). From a formal point of view, these are shorthand notations that stand for the longer form using MAKEVEC.

Interpress also provides an operator for extracting an element from a previously-constructed vector (§ 2.4.3):

⟨*v*: Vector⟩ ⟨*j*: Integer⟩ GET → ⟨*x*: Any⟩
where *x* is the value of the element of *v* named by *j*, an Integer such that $l \leq j \leq u$, where *l* is the lower bound of *v* and *u* is the upper bound.

There is no operator for "storing" values into particular elements of a Vector. Once a vector has been constructed using MAKEVEC or MAKEVECLU, its values cannot be changed. As a consequence, vectors are good for representing relatively static data structures, but are not suited to dynamic changes.

2.5 Storage mechanisms

Interpress provides several different kinds of memories that an Interpress master can use to store values. Collectively the storage areas represent the *state* of the interpreter. A programming language like Pascal provides different kinds of storage, which are defined by their names, e.g. local variables, records, static variables, and so forth. The mechanisms of the Pascal storage schemes are similar to the mechanisms of Interpress storage schemes, but the Interpress schemes are named in terms of their implementation (e.g., *the stack*) rather than in terms of their intended semantics (e.g., *local variables*).

The three storage mechanisms in Interpress are the stack, the frame, and the imager variables. They are described in turn in the following subsections.

The careful reader will remember that the Interpress base language includes the type Vector, and may wonder why the Vector does not show up in our list of storage mechanisms. The reason is that a Vector is just a value, a thing to be stored. It does indeed happen to contain storage of its own, in which further values have been arranged, but the Vector itself must be stored somewhere, and the only possible places (besides another Vector, which doesn't help) are the stack, the frame, or one of the imager variables. Recall that a Vector is read-only once it is created, so it cannot be used for general-purpose read/write storage.

2.5.1 The stack

The stack plays a central role in Interpress: all primitive operators use the stack to obtain their arguments and to return their results. We've already seen that the interpretation rules provide an automatic mechanism for placing Number, Identifier, and Body literals from the master on the stack, where they become arguments to primitive operators. To compute with any value, it must first be placed in the stack.

There are a number of operators that alter the stack contents in various ways, explained in detail in § 2.4.6. For example, POP discards the top element of the stack, EXCH will exchange the two topmost elements of the stack, and ⟨*n* COPY⟩ will copy the top *n* elements of the stack onto the stack. Since stack operators are used routinely in examples in the Introduction, you should glance at § 2.4.6 to become familiar with them.

2.5.2 The frame

The *frame* provides a means for storing local variables. Whenever a body is executed, it has access to a collection of values saved in the frame. While the stack provides a similar storage function, access to the values in the stack is generally restricted to only the top few elements. By contrast, all elements of the frame are accessible with equal ease. Although at any moment in the execution of a master there is exactly one frame accessible—hence we speak of "the frame" rather than "a frame"—there is in fact a protocol for pushing and popping frames when certain kinds of operators are executed. This complication is not important for most applications, and its further discussion is delayed until Section 12.

The frame has room for 50 distinct elements, identified by Integer indices from 0 to 49 inclusive. Actually, the Standard states that the frame has *topFrameSize* elements and requires that *topFrameSize* must be *at least* 50, so some printers may allow more than 50 elements. When the interpretation of a master begins, every frame element is set to zero. You may think of the frame as providing 50 local variables named *frame*-0 through *frame*-49, in much the same way that the Basic language provides 26 variables named A through Z. They are perfectly ordinary variables, but you get no choice in their names.

Two primitive operators are provided for accessing the frame. The FGET operator copies the value of a frame element onto the stack and the FSET operator copies a value from the stack into a frame element. Descriptions of these operators are given in § 2.4.4, but are reproduced here because later examples will make considerable use of frames:

⟨*j:* Integer⟩ FGET → ⟨*x:* Any⟩
where the value of *j* must be in the range $0 \leq j < topFrameSize$, and *x* is the current value of the frame element with index *j*.

⟨*x:* Any⟩ ⟨*j:* Integer⟩ FSET → ⟨⟩
where the value of *j* must be in the range $0 \leq j < topFrameSize$, and the value of the frame element with index *j* becomes *x*.

The FGET and FSET operators behave just like the RCL and STO (recall and store) keys on a Hewlett-Packard calculator, except that the Interpress FSET operator causes its operand to be popped from the stack, while the STO key leaves the calculator's stack intact. The following example illustrates the use of the frame to save a rather lengthy vector that will be used more than once in the master, thereby reducing the size of the master's encoding:

```
--Example 2.4--
--0--     {
--1--     <The Importance of Being Earnest> --construct a vector for a page heading--
--2--     7 FSET          --save the vector in the frame element with index 7--
-- --     --other computations would generally be included here--
--3--     7 FGET SHOW     --retrieve the heading vector and pass it as argument to SHOW--
-- --     --other computations would generally be included here--
--4--     7 FGET SHOW     --retrieve the heading vector and pass it as argument to SHOW again--
--5--     }
```

This example shows that the creator of an Interpress program must do the bookkeeping to know which value in the frame holds which variable of interest, i.e., whether it is the 7th variable in the frame or the 12th that contains a needed value.

The frame is *not* an Interpress Vector. While elements of the frame can be changed at any time, elements of an Interpress Vector cannot be changed once the vector is constructed. Thus the frame behaves like an array in a conventional programming language.

Frames are actually somewhat more complex than this description indicates. The description here remains accurate until composed operators are introduced in Section 12.

2.5.3 Imager variables

A collection of *imager variables*, sometimes simply called *variables*, is used to provide those arguments to imaging operators that are too awkward to manipulate with the stack. In effect, the imager variables hold the state of the imaging operators.

There are 23 imager variables, identified by Integer indices in the range 0 to 22 inclusive. We will defer describing the precise role of each variable. The IGET operator copies the value of an imager variable onto the stack, and the ISET operator copies a value from the stack into an imager variable (from § 4.2):

⟨*j:* Integer⟩ IGET → ⟨*x:* Any⟩
where *j* must be in the range $0 \leq j \leq 22$ and *x* is the current value of the imager variable whose index is *j*.

⟨*x:* Any⟩ ⟨*j:* Integer⟩ ISET → ⟨⟩
where *j* must be in the range $0 \leq j \leq 22$, and the value of the imager variable with index *j* becomes *x*.

It is obvious from this description that the frame and the imager variables are accessed similarly. There are important differences, however. While no primitive operator besides FSET will change a frame element, many of the imager operators change imager variables during their execution. While there is only a single set of imager variables, we shall see in a more detailed description of the base language that several different frames may exist during the execution of a master (Section 12). And finally, while the initial values of all frame elements are zero, the initial values of imager variables are chosen to be convenient.

2.6 Summary

This section has presented the most common parts of the Interpress base language, sufficient to construct a wide variety of masters. The important elements introduced are:

1. Types: Number (Integer is a subtype), Vector, Primitive Operator, Identifier, Body.
2. Literals: written forms for Number, Primitive Operator, Identifier, Body.
3. Constructors: for Vectors.
4. Interpretation rules: A master is interpreted by a stack machine that processes literals from a body sequentially. It stacks a Number, Identifier, or Body literal; it executes a Primitive Operator literal.
5. Frames: FSET and FGET access at least 50 frame elements.
6. Imager variables: ISET and IGET access 23 imager variables that play special roles in the execution of imaging operators.

The remaining constructs of the base language, namely composed operators, control operators, and marks, are described in Section 12.

3

Examples of simple masters

Before continuing with more detailed descriptions of the Interpress standard, it is time for some concrete examples of its use. All of these examples are relatively short—in fact, much shorter than most Interpress masters that are likely to be created by an application program.

All of these examples are written forms of complete, legal Interpress masters. Because Interpress is a programming language, there are many different ways to represent the same document in it. The selection of one such representation over another is primarily a matter of style, and there are fairly consistent notions of what constitutes good Interpress style. Because these are introductory examples, not all of them are examples of good Interpress style. As the sections unfold, so will our mastery of style.

3.1 A one-page line drawing

This first example produces an image containing four straight lines arranged to show the four sides of a box 1 inch high and 6½ inches wide. The bottom of the box is 9 inches from the bottom of the page and the left edge of the box is 1 inch from the left edge of the page. Assuming that the dimensions of the page are 8½ by 11 inches, the image produced will look like the one shown in Figure 3.1.

```
--Example 3.1: a simple rectangular box--
--0--     BEGIN { }       --part of the "skeleton", ignore for now--
--1--     {               --the beginning of a body that generates the page--
--2--     0.001 15 ISET   --set imager variable 15 (strokeWidth) to 0.001 --
--3--     0.0254 0.2286 0.0254 0.254   MASKVECTOR
--4--     0.1905 0.2286 0.1905 0.254   MASKVECTOR
--5--     0.0254 0.2286 0.1905 0.2286 MASKVECTOR
--6--     0.0254 0.254  0.1905 0.254   MASKVECTOR
--7--     }               --end of the page body--
--8--     END             --end of the master (more of the "skeleton")--
```

This example illustrates the overall structure of an Interpress master. Lines 0 and 8 contain literals that are part of the *skeleton*, a structure that is described in Section 4. In the example, the simplest possible skeleton is shown, configured for a master that contains only a single page to be printed. Embedded inside the skeleton is a *page body* (lines 1–7), which is interpreted by the Interpress printer in order to generate the page's image.

```
--Example 3.1: a simple rectangular box (REPRINTED FOR REFERENCE)--
--0--     BEGIN { }       --part of the "skeleton", ignore for now--
--1--     {               --the beginning of a body that generates the page--
--2--     0.001 15 ISET   --set imager variable 15 (strokeWidth) to 0.001 --
--3--     0.0254 0.2286 0.0254 0.254  MASKVECTOR
--4--     0.1905 0.2286 0.1905 0.254  MASKVECTOR
--5--     0.0254 0.2286 0.1905 0.2286 MASKVECTOR
--6--     0.0254 0.254  0.1905 0.254  MASKVECTOR
--7--     }               --end of the page body--
--8--     END             --end of the master (more of the "skeleton")--
```

In this master, distances are measured in meters. Positions on the page are measured in meters from the lower left-hand corner of the page; x values increase to the right and y values increase toward the top of the page. We shall see shortly that other conventions for measuring positions and distances can be established by invoking certain imaging operators. But if no special steps are taken, distances are measured in meters, as in this example.

Each of the four lines that form the box is drawn separately. The line at the left edge of the box is drawn by executing the literals on line 3. The first two numbers are the x and y coordinates of the line's starting point, expressed in meters (x=1 inch, y=9 inches). The second two numbers are the coordinates of the ending point of the line (x=1 inch, y=10 inches). We won't describe the MASKVECTOR operator in detail until later; for the time being, if you want to draw a line from (x_1, y_1) to (x_2, y_2), use the sequence of literals: x_1 y_1 x_2 y_2 MASKVECTOR. Literals on lines 4, 5, and 6 draw the right edge, bottom edge, and top edge of the box respectively.

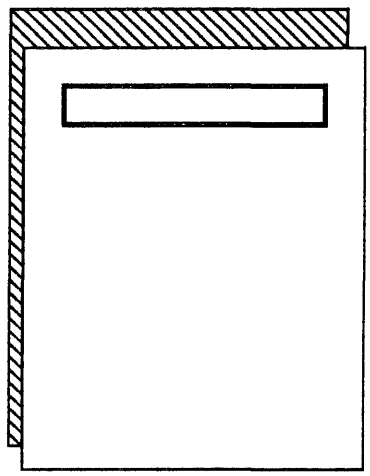


Figure 3.1. Page produced by Example 3.1

Line 2 sets an imager variable named *strokeWidth*. This variable is examined by the imaging operator MASKVECTOR to determine how wide to make the line. The example uses a stroke width of 1 millimeter (0.001 meter), although any other width could equally well be chosen. It's easy to have lines of different widths on the same image simply by insuring that *strokeWidth* is set to the desired value when MASKVECTOR is called.

This example also shows that the graphical information can be placed in the page body in any order. It is not necessary to place first those objects that will be at the top of the page, or at the left of the page, or whatever. While some printing hardware may require information to be

presented to it in a particular order, the information in the master can be in an arbitrary order. One of the jobs of Interpress printer software is to do any reordering that is required to drive a particular printing device.

You can extrapolate from this example to see how to produce an arbitrary line drawing, simply by including in the page body for each line to be drawn a set of literals similar to those on line 3. Lines can be arbitrarily short or long, at arbitrary angles. Moreover, the *strokeWidth* can be changed at any time to produce lines of different widths. It's all very straightforward.

It is instructive to observe the role of the base language in this example: it is small. The base language provides a means to specify literals and to pass them in the stack as arguments to imaging operators. In fact, this example uses only Number literals and imaging operators (ISET and MASKVECTOR). We also observe that the stack is used only to pass arguments to operators, and not for permanent storage. In fact, the stack is empty at the beginning of each line of the example.

Already in this example we begin to see the device-independent nature of Interpress. The coordinate system is based on the metric standard so as to be independent of any printer's resolution. It is also clear that the master is used to describe the appearance of an abstract image rather than to give "commands" to a printing device.

Of course, not all printers will be able to do an equally good job of making an arbitrary line drawing. Some printers may ignore any lines that are not horizontal or vertical; the box in the example will print in any case because it uses only horizontal and vertical lines, but an arbitrary line drawing may be adversely affected. Printers may also differ in their ability to make a line exactly 1 millimeter wide. Some printers may make the line 0.95 millimeters wide, some 1.13 millimeters wide, while some may be able to achieve the 1 millimeter objective almost exactly.

3.2 Simple text

The most common use of printers is certainly the printing of text. Indeed, many computer output devices sold as "printers" can do nothing else. Interpress has enormous flexibility and generality for handling text: it can print in Chinese; it can print small letters inside large letters; it can print letters in the shape of a spiral or of a mouse's tail. For this reason, its mechanisms for specifying simple text might seem somewhat baroque by comparison with those of other printers. Nevertheless, there are very simple Interpress masters that print very simple text pages.

Since Interpress printers can print in many different type fonts, a master must specify the font that is to be used. Since Interpress printers can print those type fonts in many different sizes, a master must specify the size of the letters in that font. Since Interpress can place letters anywhere at all on the page, and is not subject to any notion of "character column," a master must specify exactly where the letters will go. With these three pieces of information in hand, the master in the next example should not seem at all complex.

The Interpress master in Example 3.2 places the single word "Interpress" on one page. The word begins at a point 2.9 inches from the left of the page and 9.4 inches from the bottom of the page. The characters are in 18-point type. Figure 3.2 shows a small-scale image of the page.

```
--Example 3.2: one word of text--
--0--     BEGIN { }                --part of the "skeleton", ignore for now--
--1--     {                        --the beginning of page body--
-- --                              --the next line defines a font and saves it in frame element 0--
--2--     [ xerox, xc82-0-0, times ] FINDFONT 0.00635 SCALE MODIFYFONT 0 FSET
--3--     0 SETFONT                --sets the "current font"--
--4--     0.07366 0.23876 SETXY    --sets the "current position"--
--5--     <Interpress> SHOW        --place "Interpress" at current position in current font--
--6--     }                        --end of the page body--
--7--     END                      --end of the master (more of the "skeleton")--
```

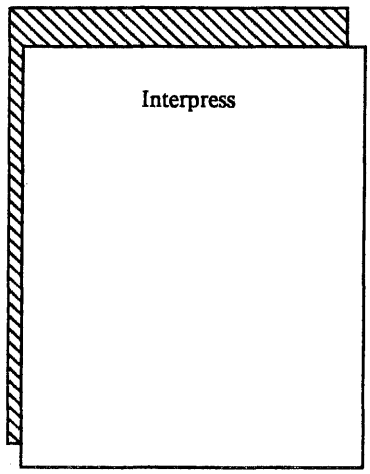


Figure 3.2. Page image produced by Example 3.2

Like Example 3.1, this example prints a single page; the page body that's embedded in the skeleton appears on lines 1–6. Lines 2 and 3 are concerned with establishing a character font and with setting up an imager variable that describes the "current font"; these activities will be discussed immediately below. Line 4 sets the "current position" to $x=2.9$ inches, $y=4$ inches, but of course these distances must be expressed in meters. Line 5 actually does the work: it "shows" the 10 characters I-n-t-e-r-p-r-e-s-s using the current font, starting at the current position. As in Example 3.1, the stack is empty at the beginning of each line.

The current position (x and y) is held in two of the imager variables (indices 0 and 1). The current position works like a cursor, identifying at all times a point on the image. The imaging operator SETXY (line 4) causes the value of these two imager variables to change; notice that unlike the example in the previous section, where we used the ISET operator to change imager variable 15 explicitly, the SETXY operator causes an implicit change to the state of the imager. Because of the ability of Interpress to handle scaling and rotation, the SETXY operator does not simply copy its arguments into the corresponding imager variables, but applies a geometric transformation to them first. For the time being, we'll assume that the current position is measured in meters and defer a discussion of transformations until Section 6.

Several imaging operators, notably SHOW, use the current position to indicate where to place something on the page image. When SHOW is called, it places an image of the first character, "I", at the current position. Then SHOW moves the current position to the right to account for the *width* of the character just shown. Then the process is repeated for the second and all subsequent characters in the character string passed as an argument to SHOW. When SHOW is finished, the current position ends up just to the right of the final "s". As each character is

shown, the current position moves to the right so as to be positioned where the immediately adjacent character should appear.

3.2.1 Setting up a font

Interpress has a comprehensive and consequently complex set of facilities for handling fonts. Because Interpress is designed to deal with high-quality typeset documents, the font machinery must be sufficiently general to deal with all of the typesetter's needs and to accommodate characters from many different printed languages. While the details of the font mechanisms must be deferred until later, enough of the basic ideas are revealed here so that lines 2 and 3 in Example 3.2 can be understood.

For each kind of character that can be placed in the page image, an Interpress printer defines a *character operator.* It is like a primitive operator in that it is by *executing* the character operator that something happens—in this case, a character is placed on the page image. But it is unlike a primitive operator in that there are no literals in the master that invoke a character operator directly. Part of the purpose of SHOW is to invoke appropriate character operators.

When a character operator is executed, it does two things:

1. It places on the page image an image of its character. The character is placed at the current position.
2. It moves the current position to the spot where the next character in a string of characters should appear.

Character operators are grouped together into collections called *fonts.* A font is a complete set of character operators, designed so that the various characters in the font appear pleasing when printed together in words and lines: their shapes have stylistic consistency and are easy to read; their sizes are consistent; the width of each character (the amount of movement of the current position) is chosen so that all possible juxtapositions of characters look pleasing; and so on. In the printing industry, a font is given a name, such as "Times Italic" or "Helvetica Light" or "Bodoni Condensed Bold." There are thousands of fonts in existence and more are being designed all the time.

In Interpress, a font is represented as a vector of character operators. The index of an operator in this vector is sometimes called its "character code." Interpress places absolutely no restrictions on the choice of character codes. As we shall see, it's up to the master's creator to arrange to invoke the proper character operators.

An Interpress printer is expected to maintain a library of fonts. A master can copy a font from the library onto the stack so that the master can make use of the character operators it defines. The FINDFONT operator is responsible for finding a font in the library and placing it on the stack (§ 4.9.1). The "name" of the font is passed to FINDFONT as a vector of identifiers. In Example 3.2, the font requested is named [*xerox, xc82-0-0, times*]. Assume for the moment that the printer has such a font in its library; we'll leave until later a discussion of all the things that could go wrong with a master's interpretation, such as a missing font. To recap, executing the following code leaves a vector of character operators on the stack:

```
[ xerox, xc82-0-0, times ] FINDFONT
```

The fonts saved in a printer's library all have a standard size: they are 1 unit high. That means that each operator is defined so that if lines of text are spaced 1 unit apart, the text will be readable. The characters in the font are thus slightly less than 1 unit high. If one of these character operators were to be executed in the page body of Example 3.2, the character would be almost a meter high! That's because the conventions in the page body are that distances are measured in meters, so a distance of 1 unit will mean a distance of 1 meter. This is clearly not what we want.

To obtain characters of the proper size, we make use of Interpress' ability to perform linear *transformations*. While the transformation features of Interpress are quite intricate, we shall use only a simple case in this example: we want to *scale* each character as it is placed on the page image. To obtain 18-point text, each character needs to be scaled so its height (originally 1 unit) becomes 18 points. (Throughout this document, we will define a point to be 1/72 inch or 0.00035278 meter. Some standards hold that there are 72.3 points to the inch. Since Interpress requires that all measurements ultimately be expressed in meters, the choice of scaling, and hence also of the size of a point, is up to the master.) To obtain an 18-point size, each character needs to be scaled by a factor of 18*0.00035278 = 0.00635. The scaling is accomplished by constructing a vector of new operators, where each new operator is a scaled version of the character operator extracted by FINDFONT. The MODIFYFONT operator, defined in § 4.9.2, will construct the new vector:

⟨*v:* Vector⟩ ⟨*m:* Transformation⟩ MODIFYFONT → ⟨*w:* Vector⟩

where *v* is a vector of operators, usually a result of FINDFONT. The result *w* is obtained from *v* by replacing each operator of *v* by an operator that applies the linear transformation *m* as its image is created. (This statement is somewhat loose; a more precise statement is given in § 4.9.2, but depends on a detailed understanding of transformations and composed operators.)

A Transformation is an Interpress type, but because there are no literals of type Transformation, values of that type must be constructed by primitive operators. The primitive operator SCALE does just what we want here (from § 4.4.3):

⟨*s:* Number⟩ SCALE → ⟨*m:* Transformation⟩

where the transformation *m* will scale *x* and *y* coordinates by a factor *s*.

So in line 2, the call to MODIFYFONT builds a vector of operators that represent an 18-point font named [*xerox, xc82-0-0, times*]. This vector is left on the stack.

The last thing that line 2 does is to save the vector of operators in the frame element with index 0. This saves the font for future reference. If we were to use several different fonts in a document, each would be saved in a different frame element.

All of the activities of line 2 are thus setup—they find and modify a font and save it in a convenient well-known place for future use.

Line 3 establishes as the "current font" the font that has been saved in frame element 0. The current font is kept in imager variable 12, named *showVec*. To set this variable, we could execute:

```
0 FGET 12 ISET
```

However, since the sequence *n* FGET 12 ISET is used quite frequently, the primitive operator SETFONT is defined so that the sequence *n* SETFONT can be used instead.

While the discussion of this section seems complex, most uses of fonts can be accomplished with a simple template. Line 2 can be taken as a template for setting up a font: finding a font of a given name, scaling each character in the font, and saving the font in a frame element. Thus the template is:

```
name FINDFONT size SCALE MODIFYFONT frameIndex FSET
```

where *name* is the name of the font, a vector of identifiers, such as [*xerox, xc82-0-0, times*]; *size* is the height of the font measured in meters (recall that the "height" is actually the minimum distance between lines of text that allows comfortable reading, sometimes called the *body size* of the characters); and *frameIndex* is the numeric index of the frame element in which to save the font. Note that this template only saves the font in the frame; it does not set the imager variable *showVec* to that font.

Once fonts have been set up with this template, we can set the current font to the one we desire with:

```
frameIndex SETFONT
```

This second template actually sets the imager variable *showVec* so that the font will be invoked with SHOW.

3.2.2 Character sets

The font machinery in Interpress makes no assumptions about the *character set* of the font, that is, the correspondence between character codes and graphic images. A character with code 14 is printed by executing the operator whose index in a font vector is 14. The SHOW operator simply extracts a character code from its argument vector and uses the numeric code to index the current font, a vector of character operators.

For example, a font might be designed so that the operator with index 1 generates an "A", the operator with index 2 a "B", and so on. If such a font were made the current font, the sequence [1, 2, 3] SHOW would print the string "ABC". However, another font might be designed so that the operator with index 65 generates an "A", the operator with index 66 a "B", and so on. When this font is the current font, the sequence [65, 66, 67] SHOW will be used to generate the string "ABC".

In our examples, we'll assume that all fonts are designed with the same conventions and that the encoding of literals written as <ABC> is consistent with these conventions. So <ABC> would encode as [1, 2, 3] if the first set of conventions is used, and as [65, 66, 67] if the second set is used.

We discuss the issue of character sets and fonts in more detail in Section 9.

3.3 An encoded example

Example 3.3 shows the encoded form of Example 3.2. If you've read § 2.5 and want to check your understanding of it, here's your chance. The written form of Example 3.2 is reprinted first, for ready reference.

```
--Example 3.2: very simple master in written form--
--0--     BEGIN { }
--1--     {
--2--     [ xerox, xc82-0-0, times ] FINDFONT 0.00635 SCALE MODIFYFONT 0 FSET
--3--     0 SETFONT
--4--     0.07366 0.23876 SETXY
--5--     <Interpress> SHOW
--6--     }
--7--     END
```

The encoding is illustrated below. Since the encoding is a binary form, we can't really show it on the page. Instead, the decimal value of each 8-bit byte is shown. The example uses the comment conventions of the other examples.

```
--Example 3.3: very simple master in encoded form--
73 110 116 101 114 112 114 101 115 115 120 88 101 114 111 120 47 50 46 48 32 --header--
160 102                              --BEGIN   (Long Op 102)--
160 106                              --{       (Long Op 106)--
160 107                              --}       (Long Op 107)--
160 106                              --{       (Long Op 106)--
197 5 120 101 114 111 120            --sequenceIdentifier 'xerox'--
197 8 120 99 56 50 45 48 45 48       --sequenceIdentifier 'xc82-0-0'--
197 5 116 105 109 101 115            --sequenceIdentifier 'times'--
15 163                               --Short Number 3--
161 27                               --Long Op 283, MAKEVEC--
160 147                              --Long Op 147, FINDFONT--
196 6 0 2 123 1 134 160              --sequenceRational n=635, d=100000--
160 164                              --Long Op 164, SCALE--
160 148                              --Long Op 148, MODIFYFONT--
15 160                               --Short Number 0--
149                                  --Short Op 21, FSET--
15 160                               --Short Number 0--
160 151                              --Long Op 151, SETFONT--
196 6 0 28 198 1 134 160             --sequenceRational n=7366, d=100000--
196 6 0 93 68 1 134 160              --sequenceRational n=23876, d=100000--
138                                  --Short Op 10, SETXY--
193 10 73 110 116 101 114 112 114 101 115 115    --sequenceString 'Interpress'--
150                                  --Short Op 22, SHOW--
160 107                              --}       (Long Op 107)--
160 103                              --END     (Long Op 103)--
```

The entire master is encoded in 112 bytes, of which 21 are in the header.

3.4 Multi-font text

Example 3.4 shows how to obtain text in several different fonts on one page. Two instances of the font setup template are used to obtain different fonts and SETFONT is then used to switch between them. Notice that the two fonts are saved in distinct frame elements.

```
--Example 3.4: text in more than one font--
-- 0--    BEGIN { }                --part of the "skeleton", ignore for now--
-- 1--    {                        --beginning of page body--
-- --                              --font 0 is 10-point (.00352778 m) Times Roman--
-- 2--    [ xerox, xc82-0-0, times ] FINDFONT 0.00352778 SCALE MODIFYFONT 0 FSET
-- --                              --font 1 is 10-point (.00352778 m) Times Italic--
-- 3--    [ xerox, xc82-0-0, timesitalic ] FINDFONT 0.00352778 SCALE MODIFYFONT 1 FSET
-- 4--    0.0508 0.254 SETXY       --sets the current position to x=2 inches, y=10 inch--
-- 5--    0 SETFONT                --use Times Roman 10 point--
-- 6--    <The > SHOW
-- 7--    1 SETFONT                --use Times Italic 10 point--
-- 8--    <Interpress Electronic Printing Standard > SHOW
-- 9--    0 SETFONT                --back to Times Roman 10 point--
--10--    <is a standard for interfacing raster> SHOW
--11--    0.0508 0.2491389 SETXY   --set current position to x=2 inch, y=(10 in)-(13 points)--
--12--    <printers to digital computers. A raster printer is an electronic device> SHOW
--13--    }                        --end of the page body--
--14--    END                      --end of the master (more of the "skeleton")--
```

This example prints two lines of text, which will appear as shown in Figure 3.3. The scale factor of Figure 3.3 is 60%. The figure is actually set in 6-point type and the 8½ inch paper width is scaled to 5.1 inches.

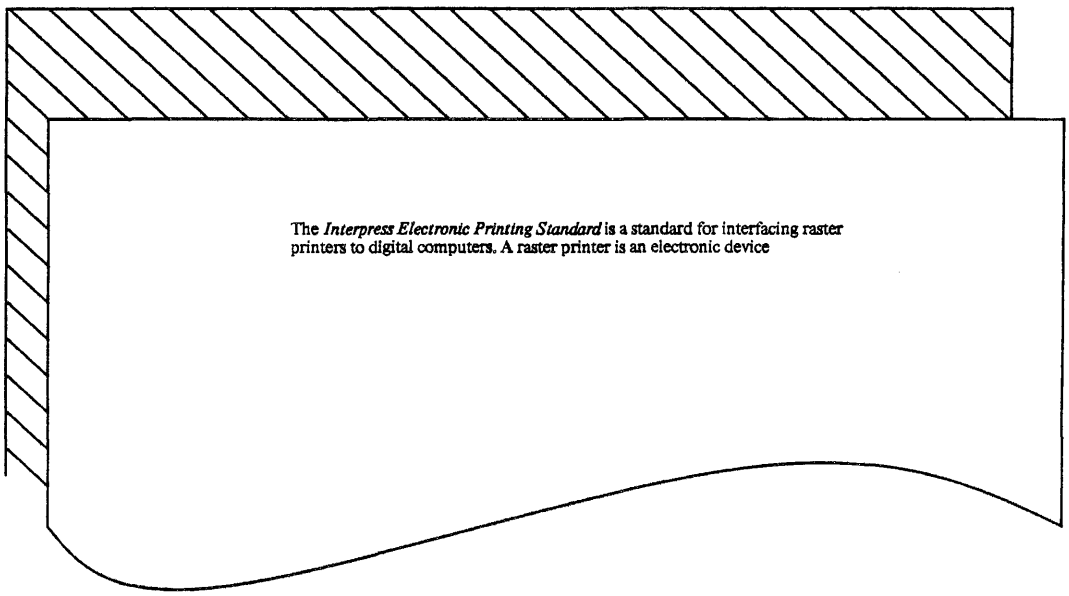


Figure 3.3. Page image produced by Example 3.4

3.5 Text and graphics

Examples 3.1 and 3.2 can be combined to produce a page that has the word "Interpress" placed within a box. The following master uses a page body that is formed by appending the page bodies of the first two examples. Figure 3.4 shows a small-scale model of the page that will be produced by this example.

```
--Example 3.5: a page with both text and graphics--
-- 0--    BEGIN { }                --part of the "skeleton", ignore for now--
-- 1--    {                        --beginning of page body--
-- 2--    0.001 15 ISET            --set imager variable 15 (strokeWidth) to 0.001 --
-- 3--    0.0254 0.2286 0.0254 0.254  MASKVECTOR
-- 4--    0.1905 0.2286 0.1905 0.254  MASKVECTOR
-- 5--    0.0254 0.2286 0.1905 0.2286 MASKVECTOR
-- 6--    0.0254 0.254  0.1905 0.254  MASKVECTOR
--  --                             --the next line defines a font and saves it in frame element 0--
-- 7--    [ xerox, xc82-0-0, times ] FINDFONT 0.00635 SCALE MODIFYFONT 0 FSET
-- 8--    0 SETFONT                --sets the "current font"--
-- 9--    0.07366 0.23876 SETXY    --sets the "current position"--
--10--    <Interpress> SHOW        --place "Interpress" at current position in current font--
--11--    }                        --end of the page body--
--12--    END                      --end of the master (more of the "skeleton")--
```

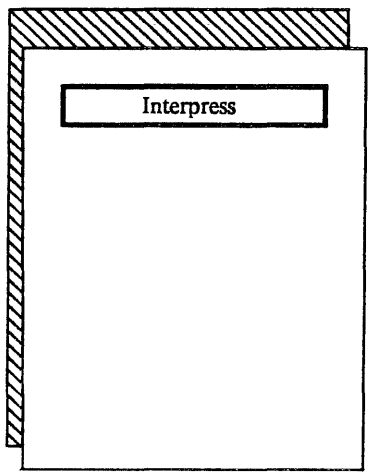


Figure 3.4. Composite page image produced by Example 3.5

As we remarked before, the order of the graphical elements within the page body is unimportant. It is important, however, that relevant imager variables contain the right values when an imaging operator is executed. In the example, the variable *strokeWidth* (index 15) is used by MASKVECTOR and SHOW uses both the current position (indices 0 and 1, set by SETXY) and the current font (index 12, *showVec*, set by SETFONT).

3.6 Multi-page documents

Suppose that instead of combining Examples 3.1 and 3.2 on one page, we wanted to make a 2-page document such that the first page looks like Example 3.1 and the second like Example 3.2. That's simple too—we put two page bodies in the master, one after another. The image built by the first page body will appear on the top page of the two-page stack produced by the printer, as shown in Figure 3.5.

```
--Example 3.6: a document with more than one page--
-- 0--    BEGIN { }                --part of the "skeleton", ignore for now--
-- 1--    {                        --beginning of the first page body--
-- 2--    0.001 15 ISET            --set imager variable 15 (strokeWidth) to 0.001 --
-- 3--    0.0254 0.2286 0.0254 0.254  MASKVECTOR
-- 4--    0.1905 0.2286 0.1905 0.254  MASKVECTOR
-- 5--    0.0254 0.2286 0.1905 0.2286 MASKVECTOR
-- 6--    0.0254 0.254  0.1905 0.254  MASKVECTOR
-- 7--    }                        --end of first the page body--
-- 8--    {                        --beginning of the second page body--
--  --                             --the next line defines a font and saves it in frame element 0--
-- 9--    [ xerox, xc82-0-0, times ] FINDFONT 0.00635 SCALE MODIFYFONT 0 FSET
--10--    0 SETFONT                --sets the "current font"--
--11--    0.07366 0.23876 SETXY    --sets the "current position"--
--12--    <Interpress> SHOW        --place "Interpress" at current position in current font--
--13--    }                        --end of the second page body--
--14--    END                      --end of the master (more of the "skeleton")--
```

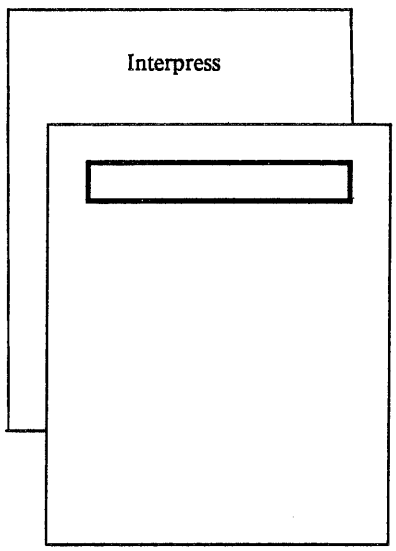


Figure 3.5. Page images produced by Example 3.6

A document with an arbitrary number of pages can be constructed simply by placing as many page bodies as needed in the skeleton.

3.7 A "line-printer listing"

A very common kind of computer output is the "listing" of a file of text. The following example shows how this might be done with Interpress, and Figure 3.6 shows the top few lines of the first two pages of the output that would be produced by this master. The figure is 60% of real size.

```
--Example 3.7: simulating a line printer listing--
-- 0--    BEGIN { }                --part of the "skeleton", ignore for now--
-- 1--    {    --beginning of the first page body; font 0 is 10-point 'LPTA'--
-- 2--    [ xerox, xc82-0-0, lpta ] FINDFONT 0.00352778 SCALE MODIFYFONT 0 FSET
-- 3--    0 SETFONT                --sets the current font--
-- 4--    0.0254 0.2667 SETXY      --heading at x=1 inch, y=10.5 inch--
-- 5--    <Listing of GP0.PAS at 14:32 on 31 January 1982       Page 1> SHOW
-- 6--    0.0254 0.254 SETXY       --top line of listing at x=1 inch, y=10 inch--
-- 7--    <1  (* GP.PAS -- Simple PASCAL graphics package. *)> SHOW
-- 8--    0.0254 0.2497667 SETXY   --next line is 12 points below first line--
-- 9--    <2  const EnterGraphicsMode=29; LeaveGraphicsMode=31;> SHOW
--10--    0.0254 0.2455333 SETXY   --each line is 12 points below previous--
--11--    <3  var xlast,ylast: integer; v: InquiryResponse;> SHOW
--12--    0.0254 0.2413000 SETXY
--13--    <4> SHOW
--14--    0.0254 0.2370667 SETXY
--15--    <5  procedure TransmitCoords(x,y: real);> SHOW
--  --    --more lines of text for the first page would be added here--
--16--    }                        --end of the first page body--
--17--    {                        --beginning of the second page body--
--18--    [ xerox, xc82-0-0, lpta ] FINDFONT 0.00352778 SCALE MODIFYFONT 0 FSET
--19--    0 SETFONT                --sets the current font--
--20--    0.0254 0.2667 SETXY --heading at x=1 inch, y=10.5 inch--
--21--    <Listing of GP0.PAS at 14:32 on 31 January 1982       Page 2> SHOW
--22--    0.0254 0.254 SETXY  --top line of listing at x=1 inch, y=10 inch--
--23--    <51  procedure DrawText(s: string);> SHOW
--  --    --more lines of text for the second page would be added here--
--24--    }                        --end of the second page body--
--  --    --more page bodies for more pages would be added here--
--25--    END                      --end of the master (more of the "skeleton")--
```

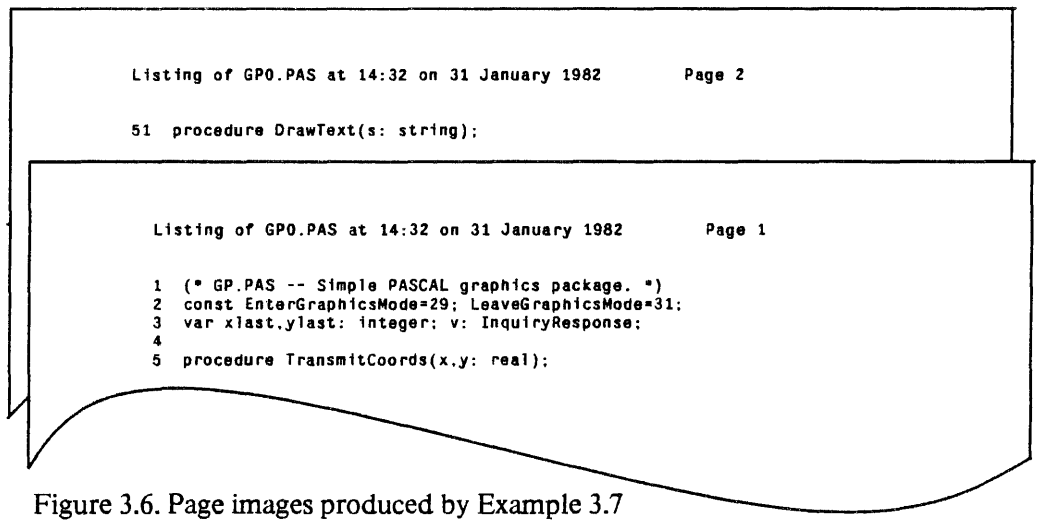


Figure 3.6. Page images produced by Example 3.7

This example illustrates an important point: the contents of the frame are not saved from one page to the next. At the beginning of each page, the frame is reset to an initial value. In the example above, therefore, the font to be used is looked up anew on each page. We'll see in Section 4 that duplicate lookups can be avoided by placing appropriate information in the master.

This example also shows that it is quite easy to prepare a master that will produce a "listing" of a text file that requires no special formatting. The listing program reads the text file and writes the encoded master in a single pass over the text. As each new line of text begins, an appropriate SETXY call is placed in the master, or a new page body is started if the text has reached the bottom of the page. Then the character codes from the text file are copied into the master, encoded as a string. (The codes can be copied provided the character set assumed in the text file is the same as the character set of the chosen font. Otherwise, character code values may have to be changed as the string is placed in the master to conform to the font's conventions.) After each string, a call to SHOW is placed in the master. It's quite a simple job, so the creator program can be very fast.

Even for the simple case of a listing, Interpress requires that the creator control formatting by preparing a master that positions every line of text. Recall that Interpress makes no formatting or typographical decisions, even in this rather simple case. Interpress printers have no facilities for "automatically" formatting or paginating a text file—these are controlled by the creator.

There are other listing formats that we might desire to use, such as rotating text 90 degrees so that it reads the long way on an 8½ by 11 inch page. Interpress can achieve this too, but the easy way to do it relies on transformations, which are treated later on in Section 6.

3.8 Summary

Most of the important parts of Interpress have been illustrated in this section. Based on the examples, you should be able to envision how to prepare a master to print almost any page, such as this one. Of course, preparing the master by hand would be impractically tedious, but a computer program could certainly do the job.

Although the masters in this section will print correctly on an Interpress printer, they depart from "good Interpress style" in three ways:

- Each page of a multi-page document duplicates font definitions. Judicious use of the *preamble*, explained in Section 4, avoids this.
- The proper use of *transformations*, explained in Sections 5 and 6, will allow masters to be expressed in units of measurement that are more convenient than meters. The masters will also be a bit more compact than the ones illustrated here.
- Every master should contain a few printing instructions. These are covered in Section 18.

4

Structure of the master

In Section 2 we explained the fundamentals of the Interpress base language and in Section 3 we showed several examples of masters. As you can see from looking at those examples, the overall structure of a master is not very complicated, but a master does have structure. Specifically, a master consists of a *preamble* followed by *page image bodies.* (We often use the shorter term *page body* interchangeably with *page image body.*) There are two primary rules governing the assembly of a master from these parts:

1. The preamble is a body that is executed before any page bodies are executed. When the preamble finishes executing, the value of the frame is saved as the *initial frame.* Fonts are usually set up in the preamble and stored in the frame.
2. Each page body is executed independently of all other page bodies. The value of the initial frame is used to initialize the frame before each page body is executed. In this way, values computed in the preamble are made available to all pages. However, modifications to the frame that are made while executing a page body cannot be detected by any other page body.

These points are elaborated in the remainder of this section and an example of the use of the preamble is given. The Standard treats these issues in § 3.

4.1 The preamble

An Interpress master consists of an arbitrary number of bodies, linked together in a *skeleton.* The skeleton may take the following form:

```
BEGIN { --preamble-- } { --page body 1-- } { --page body 2-- } --. . .-- { --page body n--} END
```

The examples in Section 3 use an empty body for the preamble and one or two page bodies. In this section, we'll explain the use of the preamble and repeat Example 3.4 using a preamble. Additional forms that the skeleton may take are deferred until Section 18.

The preamble is executed before any of the page bodies and may save computed results in the frame. When execution of the preamble is finished, the contents of the frame are saved in what is called the *page initial frame.* Thereafter, before one of the page bodies is executed, its frame is initialized to the contents of the page initial frame. In this way, the contents of the

frame at the beginning of each body are exactly equal to the contents of the frame at the end of the preamble.

This mechanism has two important implications. The first, and most obvious, is that the preamble can be used to establish values of some *global variables* whose values are saved in the frame so that they may be accessed by each page body. This feature is especially helpful for setting up fonts: the preamble contains code to find all fonts with FINDFONT, to size them with MODIFYFONT, and to save them in the frame, while calls to SETFONT in each page body are used to set the current font (the imager variable) from a frame element.

A second implication of the rule for interpreting the skeleton is that *page bodies are completely independent of one another.* A page is free to modify the frame, but these modifications cannot be detected by any other page body, because the frame's contents are set to the page initial frame as the interpretation of each page body is begun. Not only is the frame reset at the beginning of each page body, but the stack is emptied (more precisely "made to look empty"; the exact truth is given in § 3.1), and the imager variables are reset to the default values in effect when the interpretation of the master is begun. So the only way to save preamble results is in the frame. Even if each page body uses the same values for certain imager variables (e.g., current font, stroke width), these must be set explicitly at the beginning of each page body.

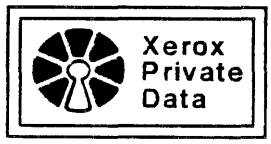
The interpretation rule thus insists that *page bodies have no side effects except to create a page image.* These rules may seem arbitrary; Section 12 explains the reasons behind them.

4.2 Examples

In practice, the preamble is used primarily to define fonts and composed operators. Since we won't talk about composed operators until Section 12, the examples in this section will use the preamble for font definitions. Examples 3.2, 3.4, 3.5, 3.6, and 3.7 could all be modified to move the font definition templates into the preamble, thus eliminating duplicates in each page body. In addition to shortening the masters, this strategy will reduce the computing the printer must do to interpret the master. Reducing the number of invocations of the FINDFONT operator is important, because this operator may have to search a data base at length to find the font requested.

The example below shows how Example 3.4 would be modified to use the preamble:

```
--Example 4.1, equivalent to 3.4--
-- 0--    BEGIN
-- 1--    {                          --beginning of the preamble--
--  --                               --font 0 is 10-point (.00352778 m) Times Roman--
-- 2--    [ xerox, xc82-0-0, times ] FINDFONT 0.00352778 SCALE MODIFYFONT 0 FSET
--  --                               --font 1 is 10-point (.00352778 m) Times Italic--
-- 3--    [ xerox, xc82-0-0, timesitalic ] FINDFONT 0.00352778 SCALE MODIFYFONT 1 FSET
-- 4--    }                          --end of the preamble--
-- 5--    {                          --beginning of page body--
-- 6--    0.0508 0.254 SETXY         --sets the current position to x=2 inch, y=10 inch--
-- 7--    0 SETFONT                  --use Times Roman 10 point--
-- 8--    <The > SHOW
-- 9--    1 SETFONT                  --use Times Italic 10 point--
--10--    <Interpress Electronic Printing Standard > SHOW
--11--    0 SETFONT                  --back to Times Roman 10 point--
--12--    <is a standard for interfacing raster> SHOW
--13--    0.0508 0.2491389 SETXY     --set current position to x=2 inch, y=(10 in)-(13 points)--
--14--    <printers to digital computers. A raster printer is an electronic device> SHOW
--15--    }                          --end of the page body--
--16--    END                        --end of the master--
```



Moving font definitions into the preamble in this example saves nothing, because the original master contains no duplicate calls to FINDFONT. However, the "line printer listing" in Example 3.7 would call FINDFONT 100 times for a 100-page listing, while FINDFONT would be called only once if the call were moved into the preamble.

4.3 Page ordering

Interpress establishes a relationship between the order of page bodies in the master and the order of the stack of printed pages. The convention is the intuitive one: the first page body produces the image that will be "on top" of the stack of images; it's the one that can be seen without opening the stack of images.

This convention may seem so obvious that it's not worth discussing. But the situation can get complicated. For example, suppose a printer is capable of printing on both sides of 8½ by 11 inch pages—what happens then? The first page body will produce the image that is on top of the stack; the second page body will produce the image on the other side of that piece of paper; the third page body will produce the top-facing image on the second piece of paper, and so on. If the printer assembles images onto pages in more complex ways, the idea is to honor the intent of the simple case: page 1 is seen first, page 2 next, and so on.

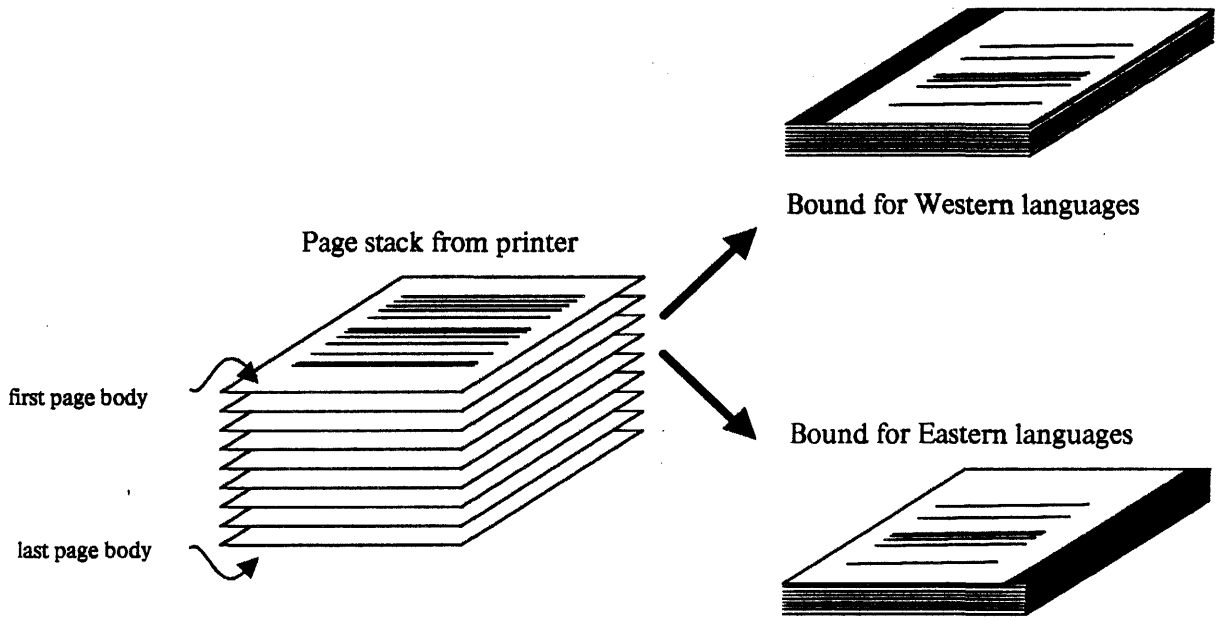


Figure 4.1. Binding determines book orientation

Some Eastern languages, such as Chinese or Farsi, are intended to be read in a manner that a Western reader often thinks of as being "backwards" or "back-to-front." In fact, the Eastern style of assembling a document differs from the Western style only in the direction that each page is to be turned when the reader is done with it, and the page orders are exactly the same. Eastern style documents are usually bound on the right, and Western style documents are usually bound on the left, but in both cases the pages in a stack are read top-to-bottom. Figure 4.1 shows this situation pictorially.

If the Interpress printer also does the binding, the situation can get more complicated. Printers designed principally for Western languages will bind only "on the left." To obtain an Eastern-like binding on such a printer, the pages are printed in the same order, but each one must be printed upside down. That is, the image on each page must be rotated 180 degrees. Techniques for preparing a master that rotates every page are given in Sections 6 and 16.

It is important to observe that the page-ordering convention is stated in terms of the stack of output, not in terms of the order in which the printer creates the images. Some printers make the last page first and grow the stack from the bottom up. Others print in the reverse order, first page first. If several copies of a document are being made, some printers will print one copy first, then the second copy, and so on, so that the pages are collated in the stack. Other printers may have mechanical sorters attached and may operate faster by making all copies of one page, then all copies of the next page, and so on. Some printers that print on both sides of the page make all front-facing images first, corresponding to bodies 1, 3, 5, etc., and then all back-facing pages, corresponding to page bodies 2, 4, 5, etc. It is precisely because the order of printing depends on details of the printing device that the execution of a page body is not allowed to have side effects: if page bodies were to have side effects, the order of their execution would be critical. Moreover, printing selected pages from a master would be cumbersome if page bodies had side effects.

One last note about the order of printing pages. When we speak of "page 1" in Interpress, we refer to the page printed by executing the first page body in the master. This page may or may not have the number "1" printed on it. It could have no visible number, it could be numbered "1", or it could be numbered "iii" or "53". Any visible page number is generated by invoking imager operators in the page body itself, and is thus controlled by the creator; Interpress never puts any image whatsoever on a page "automatically."

5

Coordinate systems

The simplest way of thinking about an Interpress master is that it describes an image by giving the precise position of every graphical object on the page. The measurement of position on the page is thus fundamental to Interpress. Whenever one measures the position of anything, it is always measured in some coordinate system. Everyone is familiar with the standard cartesian coordinate system: x to the right of the origin, and y above the origin. The basic Interpress coordinate system is just that, with the origin at the lower left corner of the page and x and y measured in meters.

Frequently it is convenient to use a distance unit other than meters, often it is convenient to use some origin besides the lower left corner of the page, and sometimes it is necessary to change the direction of x or y. Interpress therefore supports different kinds of coordinate systems and provides a way to switch back and forth from one to another. Indeed, many of the most powerful features of Interpress can be exploited only by the use and understanding of alternate coordinate systems.

5.1 Defining a coordinate system

A cartesian coordinate system is defined by an *origin* and two perpendicular *axes* beginning at the origin, used as measuring sticks. Such a coordinate system is shown in Figure 5.1.

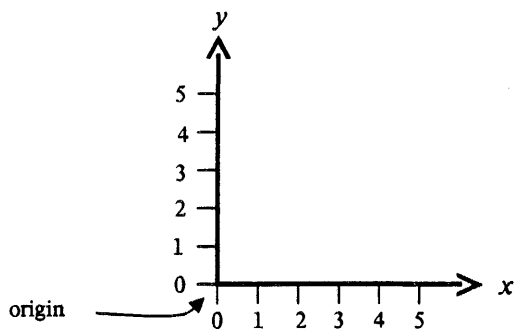


Figure 5.1. Cartesian coordinate system C.

It is usually helpful to refer to a coordinate system by a single symbol, such as *C*, which denotes the origin and axes together. In all of the coordinate systems we will use in this document, the axes are denoted by the symbols *x* and *y*. The unit of measurement, that is, the spacing of markings on the axes, is entirely arbitrary; for the time being, however, we will assume that both axes of a coordinate system have the same unit of measurement.

The location of a point is measured with respect to a coordinate system, as illustrated in Figure 5.2. The location of a point is measured as the distance from the coordinate system's origin to the point, resolved along directions parallel to the axes and measured in units defined by the axes. Thus we obtain two numbers, the *x* coordinate and the *y* coordinate; usually we write the coordinates of the point in parentheses, as (x, y). The coordinates of the point *P* in Figure 5.2 are (4, 2).

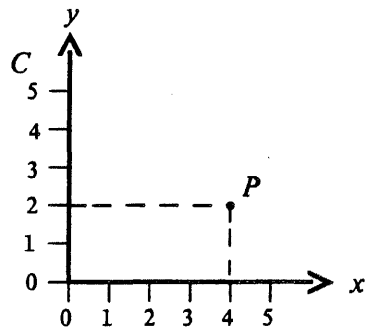


Figure 5.2. Measuring a point P with respect to C.

Since the coordinates of a point are always measured with respect to some coordinate system, we often use the name of the coordinate system as a subscript to the measured value, e.g., (x_C, y_C) or simply P_C. Note that the coordinates of a point with respect to a coordinate system are unique; that is, given numbers for *x* and *y*, the origin and the axes, one and only one point is located.

5.2 Multiple coordinate systems

The same point can be measured with respect to many different coordinate systems, resulting in different coordinate values. In Figure 5.3, two coordinate systems *A* and *B* are shown, each of which can be used to measure the coordinates of point *P*. We find that the coordinates of *P* measured with respect to *A*, P_A, are (4, 2), while the coordinates measured with respect to *B*, P_B, are (15, 9).

Since our objective in establishing coordinate systems is to describe the locations of points in an image by giving their coordinates, it may seem unnecessary to introduce the notion of multiple coordinate systems: why not establish a single standard coordinate system, and insist that all measurements be recorded with respect to that system? While this approach is sufficient, it is neither convenient nor compact. Interpress provides multiple coordinate systems so that the creator may deal in coordinate measurements that are convenient under the circumstances at hand. Multiple coordinate systems are also valuable in keeping the size of a master small, especially when printing characters.

The measurements of a point's coordinates with respect to different coordinate systems are not independent—they are in fact closely related. For example, the coordinates of the *B* and *C* systems illustrated in Figure 5.3 are related by a simple formula:

$$x_B = 2x_A + 7$$
$$y_B = 2y_A + 5$$

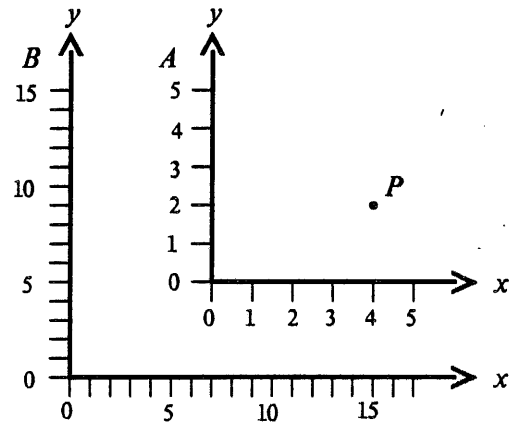


Figure 5.3. Two coordinate systems measuring one point.

Given coordinates measured in system *A*, these equations allow us to compute the coordinates of the same point measured in system *B*. Note that these conversion equations work equally well on any point. The equations thus express succinctly the relationship between the two coordinate systems *A* and *B* in a way that is independent of the points being measured.

The conversion of coordinates from one system to another is called a *coordinate transformation*. A common notation emphasizes the two coordinate systems involved: T_{AB} represents a transformation that converts coordinates measured in system *A* to coordinates measured in system *B*, and can be read as "transformation from system *A* to system *B*." While the equations given above are an example of a transformation, transformations are usually more complex and are expressed in a somewhat different form. We discuss transformations in Sections 6 and 13.

We can now glimpse the utility of transformations in Interpress. Suppose that the creator prefers to measure coordinates using system *A*, but that Interpress insists that coordinates be available in a different system *B*. For example, a creator might prefer to measure distances in printer's points from the upper left corner of the page, while Interpress internally measures distances in meters from the lower left corner of the page. Interpress permits the creator to prepare a master that expresses coordinates in system *A* and supplies a transformation T_{AB} that Interpress will apply when it is necessary to obtain measurements in system *B*.

5.3 Coordinate systems in Interpress

Although an Interpress master can be created to use any coordinate system, the definitions of the Interpress imaging operators rely on several specific coordinate systems. This section explains the reasons behind those coordinate systems and the conventions that they use.

5.3.1 The Interpress Coordinate System (ICS)

The principal coordinate system used by Interpress is a device-independent system that describes locations on the page. This coordinate system is called the *Interpress coordinate system*, or ICS, sometimes denoted by the symbol *I;* it is illustrated in Figure 5.4. The origin of the system is at the lower-left corner of the page, the *x* axis points to the right, and the *y* axis points upward. The units of distance along the axes are meters. The directions left, right, lower, and upward are defined when the page is held in the *normal viewing orientation*, described in § 4.3.1. Most often, the normal viewing orientation has the longer dimension of the page oriented vertically, as shown in Figure 5.4.

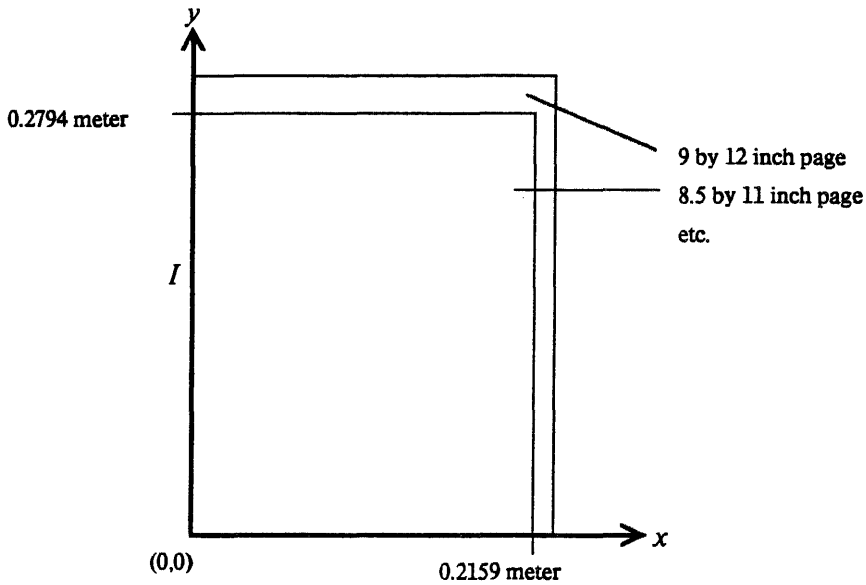


Figure 5.4. Interpress coordinate system.

The Interpress coordinate system is the coordinate system used to express locations on the page. You will recall that all of the examples in Section 3 specify coordinates in this way. We shall see in Section 6 that coordinate transformations may be specified so that the creator can express coordinates in a more convenient system and so that the printer can transform these coordinates into the ICS. But ultimately, every coordinate is converted to the ICS—this is the standard system Interpress uses internally for locating points on the page.

5.3.2 The Device Coordinate System (DCS)

The Standard describes at length the *device coordinate system*, or DCS, sometimes denoted by the symbol *D*. This coordinate system is similar to the ICS, except that the units of measurement correspond to the resolution of the printing device. Of course every Interpress printer must convert ICS coordinates into DCS coordinates in order to prepare a proper image for the printing device.

Almost all creators can ignore the device coordinate system and use only the Interpress coordinate system. In effect, a creator can imagine a fictitious printing device that measures all locations on the page in terms of the ICS; the Interpress printer software takes care of achieving this fiction.

The Standard takes care to describe the device coordinate system and operations on device coordinates in order to control precisely the effect of roundoff errors in coordinate computations. While such precise specification is required for Interpress to be a standard, most creators need not be concerned with the details. Only when extreme precision is required in an image must the properties of the DCS be understood fully.

5.3.3 The Character Coordinate System

The shape of each character in Interpress' font library is defined in a *character coordinate system*, shown in Figure 5.5. The origin of this system lies on the character's *baseline*, an imaginary horizontal line that lies just below characters such as "A" or "f". The origin is positioned at the left of the character, so that the character lies completely or nearly completely to the right of the origin. Detailed decisions about locating the origin of the character coordinate system are up to the typeface designer and may be different for foreign alphabets (see § 4.9).

The unit of measurement in the character coordinate system is the "point size" or "body size" of the character. A distance of 1 unit is the nominal (smallest) distance between lines of type of this size. This convention is chosen to match that of the printing industry: 12-point type, for example, is designed so that successive lines may be spaced 12 points (12/72 inch) apart. In the character coordinate system, then, lines of type are 1 unit apart.

All of the other geometric properties of a character are expressed in the character coordinate system. For example, the height and width of each character, as determined by the typeface designer, are measured in the character coordinate system. If a creator needs to know these dimensions, they are available in the form of *character metric information*, described in Section 9.4 and § 4.9.3.

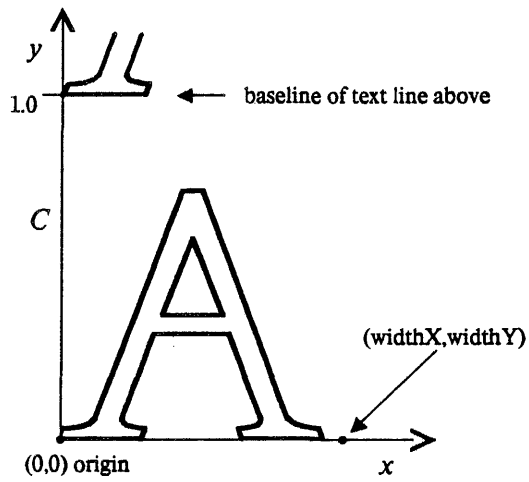


Figure 5.5. Character coordinate system.

When an image of a character is to be placed on a page, we use a transformation to specify the relationship between the character coordinate system and the Interpress coordinate system. This transformation, T_{CI}, converts coordinates of the character's definition in the font library, expressed in the character coordinate system (C), into coordinates expressed in the Interpress coordinate system (I). The transformation specifies the size, rotation, and position of the

character on the page. For each separate occurrence of the character on the page, a different transformation will be required. We'll see in the Section 6 exactly how the transformations are specified and applied.

5.3.4 Other coordinate systems

Because Interpress allows the master to contain transformation specifications, the master may define coordinate systems of its own, for its own convenience. The number and uses of these coordinate systems are limitless. They are collectively known as *master coordinate systems*, because they are coordinate systems whose conventions are determined by the master rather than by the Interpress standard.

While it is not required, a master usually sets up what we shall call a *page coordinate system*, which the master then uses to specify locations on the page. This system usually has the same origin and coordinate directions as the Interpress coordinate system, but uses a unit of distance measurement that is more convenient than the meter. For example:

- Some applications have a natural unit of measurement. For example, a typographic system might prefer the printer's *point* as the unit of measurement. Or, if this unit is not small enough to ensure sufficient accuracy, a unit of 1/10 point might be used. Of course, the creator program could convert natural units into standard ones, such as meters, but the conversion is a nuisance.
- The master may choose a coordinate system so that all coordinates can be represented as integers, leading to a more compact master. The Interpress encoding rules provide a much more compact encoding for an integer than for an arbitrary Number with a fractional part. Moreover, integers in the range -4000 to 28767 inclusive have a particularly compact Short Number encoding that requires only two bytes.

Two choices for the page coordinate system suggest themselves:

1. The unit of measurement is 1/10 point, equivalent to 1/720 inch. For an 8½ × 11 inch page, coordinates will lie in the range $0 \leq x \leq 6120$ and $0 \leq y \leq 7920$. A number within this range can be specified with the Short Number encoding, leading to a compact master.
2. The unit of measurement is 10^{-5} meter, a unit sometimes called the *mica*. For an 8½ × 11 inch page, coordinates will lie in the range $0 \leq x \leq 21590$ and $0 \leq y \leq 27940$. Because the mica is smaller than 1/10 point, the mica system has somewhat more precision.

If a master uses either of these systems, or any other page coordinate system that differs from the Interpress coordinate system, the master will need to specify a transformation that converts from the system it uses into the Interpress coordinate system.

5.4 Summary

This section has described Interpress' use of coordinate systems. Some of these systems are standardized by Interpress, such as the Interpress coordinate system (ICS) and the character coordinate system. Others, not standardized, can be established by the master, using operators of the Interpress language. A discussion of coordinate systems is not complete without describing mechanisms for transforming coordinates measured in one system to coordinates measured in another system. The following section describes transformation mechanisms and presents several examples of their use.

6

Transformations I

A coordinate transformation is used to convert coordinates measured in one coordinate system into coordinates measured in another system. Interpress uses transformations extensively to help position objects on the page. This section provides an introduction to transformations.

There are two major parts to the story of transformations: how they are constructed and how they are used. While a transformation is a type in the Interpress language, transformations cannot be included in masters as literals, but must instead be constructed by calling one of a number of imaging operators that build transformations.

During the execution of a master there is always a *current transformation*, which is applied to all coordinates specified in the master. A master that measures coordinates in inches must create a transformation that converts inches to meters and then incorporate it into the current transformation. In Section 6.3 we explain the concept of combining transformations and explain that a master never actually replaces the current transformation, but merely adds to it.

Throughout this section and the rest of this document, we shall use a common notation for transformations. The symbol T, without subscript, refers to the current transformation, which is held as the value of imager variable 4. Other transformations use the symbol T with subscripts, as in T_{ab}. The subscripts denote the coordinate systems involved, i.e., the transformation T_{ab} converts coordinates measured in system a into coordinates measured in system b.

6.1 What is a transformation?

Mathematically, a transformation is a function that inputs an (x, y) pair in one coordinate system and outputs the equivalent values in another coordinate system. In Interpress, a Transformation is a data type, just as Vector and Number are types. But unlike those data types, a Transformation is intended to be used as a function and not as a datum. In use, a Transformation accepts coordinate pairs and delivers transformed coordinate pairs.

Since the output of a transformation (a coordinate pair) is the same kind of data as the input to a transformation (a coordinate pair), it is possible to make compound transformations the same way that we make compound lenses. A microscope, for example, may achieve a 600-power magnification with a 60-power objective lens and a 10-power eyepiece; we look through both of them at the same time. In a camera, we can add a 2X tele-extender to a 300-mm lens

and achieve the equivalent of a 600-mm lens. An optician, while fitting you for eyeglasses, can put a 2-diopter lens, a 0.1-diopter lens, and a 90-degree astigmatism corrector all in a line in front of your eye to achieve the equivalent of an eyeglass lens that has 2.1 diopters of correction and a 90-degree astigmatism axis.

While you may not be intimately familiar with all these various uses of optics, the point is that a wide variety of effective lenses can be made by combining component lenses and using them together as a single instrument. Transformations work in pretty much the same way: with a small number of building-block transformations called *primitive transformations*, we can build anything we want.

6.2 Constructing transformations

There are two ways to construct transformations. First, there are several operators that will construct *primitive transformations*, based on arguments passed to these operators. The operators leave the resulting transformation on the stack. A primitive transformation may specify a scaling, a translation, or a rotation. Some transformations cannot be expressed as a primitive transformation, but can be expressed in a second way: a transformation can be created by combining two existing transformations. An arbitrary transformation can be constructed by combining primitive transformations in various ways.

6.2.1 Primitive transformations

There are three kinds of primitive transformations: scaling, translation, and rotation. Each transformation is constructed with one of the operators: SCALE, TRANSLATE, and ROTATE (§ 4.4.3).

A scaling transformation is built with the SCALE operator, which takes a single argument:

⟨*s*: Number⟩ SCALE → ⟨T_{ft}: Transformation⟩

where T_{ft} is a transformation that will convert coordinates measured in the *f* system into those measured in the *t* system. The two systems have the same origin and axes pointing in the same directions, but the units of distance measurement are different: 1 unit in the *f* system corresponds to *s* units in the *t* system.

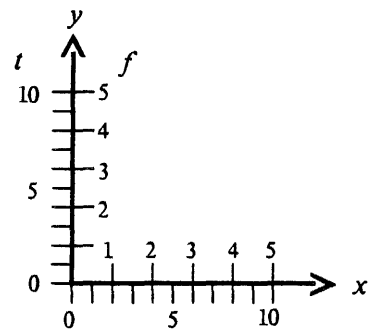


Figure 6.1. Primitive scaling transformation.

Figure 6.1 illustrates two coordinate systems related by a scaling transformation with $s=2$. The illustration is a bit awkward because the axes of the two systems lie over one another. In this case, we adopt the convention of placing labels for the units of one coordinate system on one side of the axis line and labels for the other system on the other side.

A translation transformation describes the relation between two coordinate systems that differ only in the position of the origin:

⟨t_x: Number⟩ ⟨t_y: Number⟩ TRANSLATE → ⟨T_{ft}: Transformation⟩

where T_{ft} is a transformation that converts coordinates measured in the *f* system into those measured in the *t* system. The two systems have axes pointed the same direction and measured in the same units, but have different origins. The origin in the *f* system corresponds to the point (t_x, t_y) in the *t* system.

Figure 6.2 illustrates two coordinate systems related by a translation with $t_x=3$, $t_y=2$.

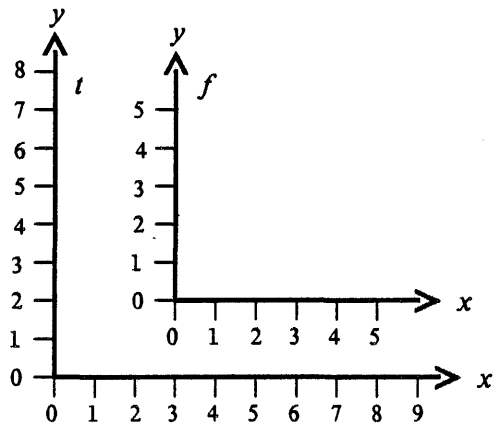


Figure 6.2. Primitive translation transformation.

Finally, a rotation transformation relates two coordinate systems that differ only by a rotation of axes about the origin:

⟨*a*: Number⟩ ROTATE → ⟨T_{ft}: Transformation⟩

where T_{ft} is a transformation that converts coordinates measured in the *f* system into those measured in the *t* system. The two systems have the same origin and units of measurement, but have axes rotated by an angle *a*, measured in degrees. The angle is measured clockwise from an axis of the *f* system to the corresponding axis of the *t* system.

Figure 6.3 illustrates two coordinate systems related by a rotation with $a=30$ degrees.

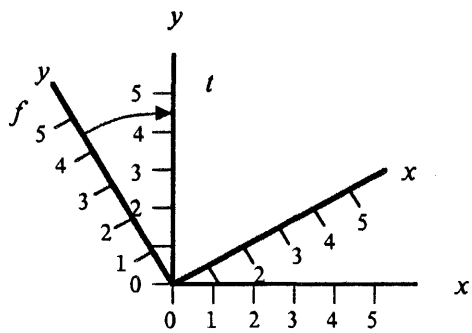


Figure 6.3. Primitive rotation transformation.

6.2.2 Combining transformations

Complex transformations are built up by combining, or *concatenating*, simpler transformations. Suppose we have a transformation T_{ab} that converts coordinates from system a to system b, and a second transformation T_{bc} that converts from system b to system c. If we wanted to convert a coordinate measured in system a to a measurement in system c, it might appear that we must first apply T_{ab} and then apply T_{bc} to the results. However, it turns out that the two transformations can be concatenated into a single transformation T_{ac} that accomplishes just this effect. The concatenation is performed by the CONCAT operator (§ 4.4.3):

$\langle T_{ab}$: Transformation$\rangle$ $\langle T_{bc}$: Transformation$\rangle$ CONCAT → $\langle T_{ac}$: Transformation$\rangle$
where T_{ac} is a transformation that has the same effect as first applying T_{ab} and then applying T_{bc}.

We shall see below that masters use CONCAT a great deal, both implicitly and explicitly, to combine transformations.

When transformations are concatenated, sequence is important. Applying T_{ab} and then T_{bc} will not, in general, yield the same result as applying T_{bc} and then T_{ab}. In mathematical terms, we say that the operation of concatenating transformations is *not commutative*. Intuitively, the reason for this is that rotational transformations always pivot around the origin of the coordinate system and not the center of the object being rotated, so that if we translate an object before we rotate it, the rotation will swing it around like a weight at the end of a string rather than just spinning it in place. We discuss this issue some more, with a bit of mathematical rigor, in Section 13.

Since the order of concatenation is important, we will always write concatenations in such a way that they may be easily read from left to right. That is, $\langle T_{ab}\ T_{bc}$ CONCAT$\rangle$ can be thought of as *first* applying transformation T_{ab} and *then* applying T_{bc}.

Sometimes a string of concatenated transformations can get quite long. Suppose we want to achieve the effect of applying T_{ab}, then T_{bc}, then T_{cd}, then T_{de}, a sequence which we might write directly as $T_{ab}\ T_{bc}\ T_{cd}\ T_{de}$. These can all be concatenated together by $\langle T_{ab}\ T_{bc}\ T_{cd}\ T_{de}$ CONCAT CONCAT CONCAT$\rangle$. There is no need to place all the CONCAT operators at the end; we could just as well write $\langle T_{ab}\ T_{bc}$ CONCAT T_{cd} CONCAT T_{de} CONCAT$\rangle$.

It often happens that the current transformation is to be modified by concatenating to it a transformation that will be performed first. That is, if T is the current transformation, we want to replace it with a transformation $T_n\ T$. While we could write $\langle T_n$ 4 IGET CONCAT 4 ISET$\rangle$, since the current transformation is the value of imager variable number 4, the operator CONCATT (note two T's) is available to do this job (§ 4.4.5):

$\langle T_n$: Transformation$\rangle$ CONCATT → $\langle\rangle$
where T is set to $\langle T_n\ T$ CONCAT$\rangle$.

It is useful if you think of the current transformation defining a "current coordinate system." At the beginning of a page body, the current coordinate system will always be the Interpress coordinate system, *I*. Coordinates in the master that are subjected to the current transformation will thus be expressed in system *I*. If the current transformation is modified within the page body, a new coordinate system is defined, and coordinates will be interpreted in the new sys-

tem. For example, suppose you have constructed a transformation T_{PI} that converts coordinates from system *P* to system *I*. If the current transformation is altered by executing ‹T_{PI} CONCATT›, the current coordinate system becomes *P*, so coordinates in the master must be expressed in that system.

6.2.3 Examples

Now that we've developed mechanisms to build up an arbitrary transformation from primitive building-block transformations, it's time to verify our understanding with some examples.

First, let's consider the example shown in Figure 5.3, which is repeated in Figure 6.4 below. We wish to formulate a transformation T_{AB} that will convert coordinates measured in the *A* system into coordinates measured in the *B* system. This transformation cannot be expressed as a primitive transformation, because both origins and scales differ. Thus both a scaling and a translation will be required.

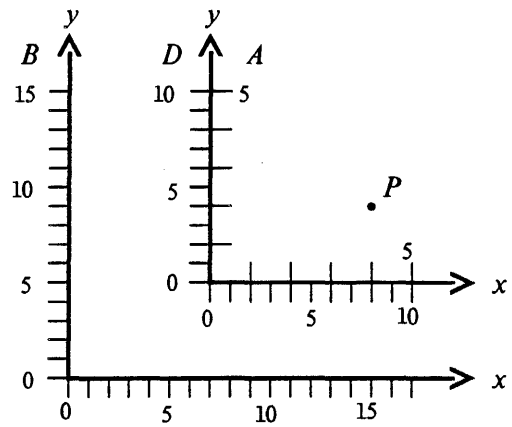


Figure 6.4. Using system D to help transform from A to B

Let us define another coordinate system *D*, which has the origin and axis directions of system *A* but the scale of system *B*. This system is shown in Figure 6.4, but is hard to see because its axes fall over those of system *A*. Now we see that systems *A* and *D* are related by a primitive scaling transformation and systems *D* and *B* are related by a primitive translation. By inspection, we determine: T_{AD} = ‹2 SCALE› and T_{DB} = ‹7 5 TRANSLATE›. The complete transformation T_{AB}, which has the effect of applying T_{AD} and then T_{DB}, is thus obtained by ‹2 SCALE 7 5 TRANSLATE CONCAT›. Note that this is *not* the same transformation if we perform the primitive transformations in the opposite order, i.e., ‹7 5 TRANSLATE 2 SCALE CONCAT›.

A second example is illustrated in Figure 6.5. The overall objective is to convert from system *C* to system *I;* systems *S* and *P* are used for other purposes. The transformations involved can be determined by inspecting the illustration: T_{CS} = ‹353 SCALE›, T_{SP}= ‹700 400 TRANSLATE›, and T_{PI} = ‹0.00001 SCALE›. Thus the complete transformation T_{CI}, should we need to express it as a single transformation, is ‹353 SCALE 700 400 TRANSLATE 0.00001 SCALE CONCAT CONCAT›. There is no reason to save the CONCAT operators for last; we could just as well write ‹353 SCALE 700 400 TRANSLATE CONCAT 0.00001 SCALE CONCAT›.

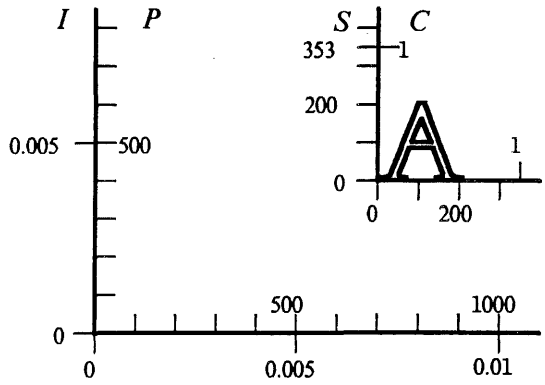


Figure 6.5. Coordinate systems for printing a character.

This example illustrates precisely how characters defined in the character coordinate system *C* are transformed into the Interpress coordinate system *I*. While systems *S* and *P* have no formal names in Interpress, we shall name them the *scaled character coordinate system* and the *page coordinate system*. While Interpress does not insist that these systems be used in the way illustrated by Figure 6.5, "good Interpress style" suggests their use.

The page coordinate system is chosen by the creator as a convenient one for expressing locations on the page; the choice was discussed in Section 5.3.4. In the example in Figure 6.5, the page coordinate system is expressed in units of micas, 10^{-5} meter. The master establishes the page coordinate system by invoking the CONCATT operator at the beginning of each page body to modify the current transformation. This coordinate system will stay in effect throughout the page, unless of course the master invokes operators that change the current transformation again. Section 6.3.1, below, shows an example of how this is done.

The scaled character coordinate system *S* has the same scale and orientation as the page coordinate system; in this way, a character expressed in the *S* system is located with respect to the page system by a primitive translation transformation T_{SP}. This translation transformation is provided by the SHOW operator, which uses the current position to determine where the origin of the scaled coordinate system should be placed. The formal definition of SHOW (§ 4.4.6) describes how T_{SP} is formed and concatenated with *T* so that the current coordinate system becomes *S*.

The transformation T_{CS} is the transformation given as an argument to MODIFYFONT. The MODIFYFONT operator creates a set of character operators that will first concatenate T_{CS} onto *T* so that the current coordinate system becomes *C*. Now a standard character definition in the font library, which is expressed using coordinates measured in system *C*, can be used to describe the shape of the character.

All the information governing a particular occurrence of a character in an image is contained in the transformations: T_{CS} and T_{PI} determine the size of the character, and T_{SP} and T_{PI} determine its location.

Table 6.1 reviews the state of the current transformation as the process of printing a character unfolds.

Table 6.1 Transformations involved in printing a character

Current transformation	Current coordinate system	Operation performed
T_{ID}	*I*	At beginning of a page body, the printer sets the current transformation to convert from Interpress coordinates (ICS) to device coordinates (DCS).
		⟨T_{PI} CONCATT⟩ is performed by the master as its first action in the page body.
$T_{PI}T_{ID}$	*P*	The "page coordinate system" is established.
		⟨T_{SP} CONCATT⟩ is performed as part of the SHOW operator, using the current position as an argument (actually TRANS does the work—§ 4.4.5).
$T_{SP}T_{PI}T_{ID}$	*S*	The "scaled character coordinate system" is established.
		⟨T_{CS} CONCATT⟩ is performed by the character operator created by MODIFYFONT.
$T_{CS}T_{SP}T_{PI}T_{ID}$	*C*	The standard character coordinate system.

It appears from this sequence that the current transformation is altered properly to print the character in Figure 6.5, but that it remains set for the *C* system for that particular character. In fact, when the SHOW operator finishes execution, it restores *T* to the value it had when SHOW was entered. In this way, the page coordinate system is once again the current coordinate system. The only side effect of SHOW, in addition to placing character images on the page, is to alter the current position to account for the width of the characters imaged.

It is worth emphasizing that the *ultimate size and rotation of a character is determined both by the current transformation in effect when* SHOW *is called and by the transformation given to* MODIFYFONT *to establish the font.*

6.3 Using transformations

The facility with coordinate systems and transformations that we have developed in the last few sections now allows us to apply "good Interpress style" to the examples given in Section 3.

6.3.1 Simple page coordinate system

By using the notion of a page coordinate system, a master will become more compact because numbers can be encoded with sufficient precision as 2-byte Short Numbers (§ 2.5.1) rather

than as rationals, the equivalent of floating-point numbers. All that is required to establish a page coordinate system is to place first, in each page body, a piece of Interpress program that constructs an appropriate T_{PI} transformation and concatenates it onto T.

Example 3.1 is given below, modified to use a page coordinate system that uses units of 10^{-5} meter (micas) rather than the meter units of the Interpress coordinate system. While Example 3.1 requires 146 bytes to encode, Example 6.1 requires only 90. The savings will be even greater in more complex masters, which are common.

```
--Example 6.1. Produces the same image as Example 3.1--
--0--     BEGIN { }                  --empty preamble--
--1--     {                          --the beginning of a body that generates the page--
-- --                                --set the page coordinate system by concatenating onto T--
--2--     0.00001 SCALE CONCATT      --TPI = <0.00001 SCALE>--
--3--     100 15 ISET                --set imager variable 15 (strokeWidth) to 100 micas --
--4--      2540 22860  2540 25400 MASKVECTOR
--5--     19050 22860 19050 25400 MASKVECTOR
--6--      2540 22860 19050 22860 MASKVECTOR
--7--      2540 25400 19050 25400 MASKVECTOR
--8--     }                          --end of the page body--
--9--     END                        --end of the master--
```

Example 3.2 is given below, again using a page coordinate system in units of 10^{-5} meter. Note that in line 3, the transformation passed to MODIFYFONT has been changed so that T_{CS} will scale the character so that the unit distance in the standard character coordinate system will correspond to 635 units in the scaled character coordinate system; these units correspond to the page coordinate system (635×10^{-5} meter = 18 points).

```
--Example 6.2. Produces the same image as Example 3.2--
--0--     BEGIN { }                  --empty preamble--
--1--     {                          --beginning of the page body--
-- --                                --set the page coordinate system by concatenating onto T--
--2--     0.00001 SCALE CONCATT      --TPI = <0.00001 SCALE>--
-- --                                --define a font and save it in frame element 0--
--3--     [ xerox, xc82-0-0, times ] FINDFONT 635 SCALE MODIFYFONT 0 FSET --TCS = <635 SCALE>--
--4--     0 SETFONT                  --set the "current font"--
--6--     7366 23876 SETXY           --set the "current position"--
-- --                                --TSP = <7366 23876 TRANSLATE>--
--6--     <Interpress> SHOW          --place "Interpress" at current position in current font--
--7--     }                          --end of the page body--
--8--     END                        --end of the master--
```

The next example shows Example 4.1 (derived from Example 3.4) reworked to use a page coordinate system expressed in points (0.00035278 meter). Note that the calls to MODIFYFONT in the preamble have been changed so that the scaling transformation will convert from the standard character coordinate system to the units of the page coordinate system (points).

```
--Example 6.3. Produces the same image as Examples 4.1 and 3.4--
-- 0--    BEGIN
-- 1--    {                          --beginning of the preamble--
--  --                               --font 0 is 10-point Times Roman--
-- 2--    [ xerox, xc82-0-0, times ] FINDFONT 10 SCALE MODIFYFONT 0 FSET
--  --                               --font 1 is 10-point Times Italic--
-- 3--    [ xerox, xc82-0-0, timesitalic ] FINDFONT 10 SCALE MODIFYFONT 1 FSET
-- 4--    }                          --end of the preamble--
-- 5--    {                          --beginning of page body--
-- 6--    0.00035278 SCALE CONCATT --set the page coordinate system by concatenating onto T--
-- 7--    144 720 SETXY              --set the current position to x=2 inch, y=10 inch--
-- 8--    0 SETFONT                  --use Times Roman 10 point--
-- 9--    <The > SHOW
--10--    1 SETFONT                  --use Times Italic 10 point--
--11--    <Interpress Electronic Printing Standard > SHOW
--12--    0 SETFONT                  --back to Times Roman 10 point--
```

```
--13--    <is a standard for interfacing raster> SHOW
--14--    144 707 SETXY              --set current position to x=2 inch, y=(10 in)-(13 points)--
--15--    <printers to digital computers. A raster printer is an electronic device> SHOW
--16--    }                          --end of the page body--
--17--    END                        --end of the master--
```

As a final example, let's redo Example 3.7 to use a page coordinate system based on units of 1/10 point (0.000035278 meter) and to place the font definition in the preamble so that it is not duplicated in each page body.

```
--Example 6.4. Produces the same image as Example 3.7--
-- 0--    BEGIN
-- 1--    {                          --beginning of the preamble--
-- --                                --font 0 is 10-point 'LPTA'--
-- 2--    [ xerox, xc82-0-0, lpta ] FINDFONT 100 SCALE MODIFYFONT 0 FSET
-- 3--    }                          --end of the preamble--
-- 4--    {                          --beginning of the first page body--
-- 5--    0.000035278 SCALE CONCATT --set the page coordinate system by concatenating onto T--
-- 6--    0 SETFONT                  --sets the current font--
-- 7--    720 7560 SETXY             --heading at x=1 inch, y=10.5 inch--
-- 8--    <Listing of GP0.PAS at 14:32 on 31 January 1982  Page 1> SHOW
-- 9--    720 7200 SETXY             --top line of listing at x=1 inch, y=10 inch--
--10--    <1  (* GP.PAS -- Simple PASCAL graphics package. *)> SHOW
--11--    720 7080 SETXY             --next line is 12 points below first line--
--12--    <2  const EnterGraphicsMode=29; LeaveGraphicsMode=31;> SHOW
--13--    720 6960 SETXY             --each line is 12 points below previous--
--14--    <3  var xlast,ylast: integer; v: InquiryResponse;> SHOW
--15--    720 6840 SETXY
--16--    <4> SHOW
--17--    720 6720 SETXY
--18--    <5  procedure TransmitCoords(x,y: real);> SHOW
-- --     --more lines of text for the first page would be added here--
--19--    }                          --end of the first page body--
--20--    {                          --beginning of the second page body--
--21--    0.000035278 SCALE CONCATT --set the page coordinate system by concatenating onto T--
--22--    0 SETFONT                  --sets the current font--
--23--    720 7560 SETXY --heading at x=1 inch, y=10.5 inch--
--24--    <Listing of GP0.PAS at 14:32 on 31 January 1982  Page 2> SHOW
--25--    720 7200 SETXY             --top line of listing at x=1 inch, y=10 inch--
--26--    <51  procedure DrawText(s: string);> SHOW
-- --     --more lines of text for the second page would be added here--
--27--    }                          --end of the second page body--
-- --     --more page bodies for more pages would be added here--
--28--    END                        --end of the master--
```

Note that the page coordinate system must be established for each page body. This action is required because the imager variables, including the current transformation T, are reset at the beginning of each page body to the default values in effect when the interpretation of the master is begun. In this respect, the current transformation is like the current font—both CONCATT and SETFONT must be called within each page body to set the imager variables appropriately. (See Section 12 for more discussion of why this is so.)

A slight improvement in Example 6.4 could be achieved by computing the transformation T_{PI} in the preamble and saving it in a frame element, similar to the way a font is saved:

```
--2.5--   0.000035278 SCALE 1 FSET --compute transformation and save in frame element 1--
```

Within each page body, the page coordinate system can be established by concatenating this transformation onto T. The following line would replace lines 5 and 21 in Example 6.4:

```
-- --     1 FGET CONCATT --set the page coordinate system by concatenating onto T--
```

6.3.2 Landscape page coordinate system

All of the examples so far have placed text on the page without rotation—that is, the baseline of the text is aligned with the x axis of the Interpress coordinate system. By using appropriate transformations, text with arbitrary rotations may be placed on the page.

The most frequent use for rotated images is when the entire page is to be rotated, sometimes called a *turned page* or a *landscape page.* Figure 6.6 shows such a page, in which text starts at the bottom and runs up the page.

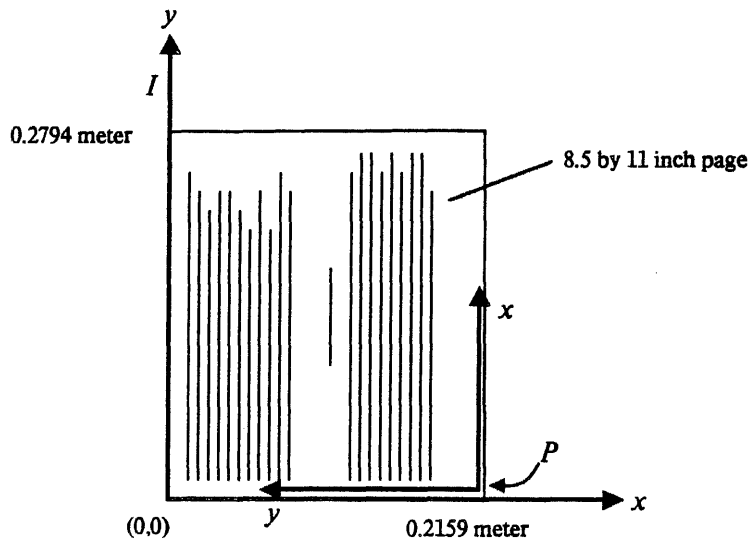


Figure 6.6. Coordinate system for landscape printing.

The easiest way to prepare a master for a landscape page is to formulate a page coordinate system in which the x axis is aligned with the character baseline. Such a coordinate system P is illustrated in Figure 6.6: the origin is at the lower right corner of the page, the y axis points to the left, and the x axis points up the page. If we rotate the page 90 degrees clockwise, we see that these conventions correspond exactly to the Interpress coordinate system on an unrotated page. Of course, the page coordinate system we use can establish a convenient unit of measurement as well as the proper origin and axis directions.

The important point about establishing a landscape page coordinate system is that the systems used in the imaging of characters (systems S and C described above) are expressed relative to the page coordinate system. So simply rotating the page coordinate system will rotate all information on the page.

Let us now work out how to express T_{PI}, the transformation from the page coordinate system illustrated in Figure 6.6 to the Interpress coordinate system. It helps if we envision two other coordinate systems, A and B, as illustrated in Figure 6.7.

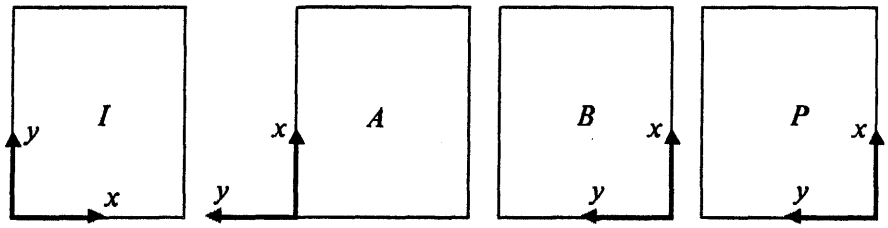


Figure 6.7. Steps in deriving the landscape transformation.

System *A* is simply a rotated version of the Interpress coordinate system, so T_{AI} = ⟨90 ROTATE⟩. The rule for obtaining the angle says that we measure the angle clockwise from an axis of the *A* system to the corresponding axis of the *I* system.

System *B* is system *A* translated so its origin will be at the bottom right corner of the page. Thus T_{BA} = ⟨0 −0.2159 TRANSLATE⟩, where the width of the page is 8.5 inches, or 0.2159 meters. Note that the translation is along the *y* axis.

Finally, system *P* is system *B* scaled so that units of measurement are convenient. In this example, we'll assume that units of 1/10 point (0.000035278 meter) are to be used. Hence T_{PB} = ⟨0.000035278 SCALE⟩.

The complete transformation T_{PI} is obtained by concatenating T_{PB}, T_{BA}, and T_{AI}, in that order. Thus to place on the stack the transformation T_{PI}, we could execute:

```
0.000035278 SCALE 0 -0.2159 TRANSLATE CONCAT 90 ROTATE CONCAT
```

To establish the page coordinate system *P* as the current coordinate system, we would prepare the transformation T_{PI} and then execute CONCATT, provided the coordinate system previously in effect was the Interpress coordinate system, *I*.

Example 6.5 illustrates how a landscape listing can be produced. The content of the listing is the same as in Examples 6.4 and 3.7, but the text will now be read along the long axis of the paper. Note that the coordinates of the starting of text lines have been changed to reflect the fact that the *y* axis of the page coordinate system has a useful range of 8.5 rather than 11 inches. Note too that since the page-to-Interpress transformation is a complex one, it is computed once in the preamble and concatenated onto *T* in each page body.

```
--Example 6.5.--
-- 0--    BEGIN
-- 1--    {                          --beginning of the preamble--
--  --                               --font 0 is 10-point 'LPTA'--
-- 2--    [ xerox, xc82-0-0, lpta ] FINDFONT 100 SCALE MODIFYFONT 0 FSET
--  --                               --following line computes page-to-Interpress transformation--
-- 3--    0.000035278 SCALE 0 -0.2159 TRANSLATE CONCAT 90 ROTATE CONCAT 1 FSET
-- 4--    }                          --end of the preamble--
-- 5--    {                          --beginning of the first page body--
-- 6--    1 FGET CONCATT             --set the page coordinate system by concatenating onto T--
-- 7--    0 SETFONT                  --sets the current font--
-- 8--    720 5760 SETXY             --heading at x=1 inch, y=8 inch--
-- 9--    <Listing of GP0.PAS at 14:32 on 31 January 1982  Page 1> SHOW
--10--    720 5400 SETXY             --top line of listing at x=1 inch, y=7.5 inch--
--11--    <1  (* GP.PAS -- Simple PASCAL graphics package. *)> SHOW
--12--    720 5280 SETXY             --next line is 12 points below first line--
```

```
--13--    <2  const EnterGraphicsMode=29; LeaveGraphicsMode=31;> SHOW
--14--    720 5160 SETXY              --each line is 12 points below previous--
--15--    <3  var xlast,ylast: integer; v: InquiryResponse;> SHOW
--16--    720 5040 SETXY
--17--    <4> SHOW
--18--    720 4920 SETXY
--19--    <5  procedure TransmitCoords(x,y: real);> SHOW
--  --    --more lines of text for the first page would be added here--
--20--    }                           --end of the first page body--
--21--    {                           --beginning of the second page body--
--22--    1 FGET CONCATT              --set the page coordinate system by concatenating onto T--
--23--    0 SETFONT                   --sets the current font--
--24--    720 5760 SETXY --heading at x=1 inch, y=8 inch--
--25--    <Listing of GP0.PAS at 14:32 on 31 January 1982  Page 2> SHOW
--26--    720 5400 SETXY              --top line of listing at x=1 inch, y=7.5 inch--
--27--    <51  procedure DrawText(s: string);> SHOW
--  --    --more lines of text for the second page would be added here--
--28--    }                           --end of the second page body--
--  --    --more page bodies for more pages would be added here--
--29--    END                         --end of the master--
```

6.3.3 Multiple page coordinate systems

The current transformation can be manipulated to establish a convenient coordinate system at any point during the preparation of a page. Because the current transformation, *T*, is an imager variable, it may be saved, modified, and restored at will.

For example, suppose that you want to make a landscape listing of some text, but that the heading is to read properly when the paper is held in the normal viewing orientation. The page coordinate system used in Example 6.5 is appropriate for the text, while the page coordinate system used in Example 6.4 is appropriate for the heading. The following sketch of a master shows how the current transformation can be manipulated for this purpose:

```
--Example 6.6.--
-- 0--    BEGIN
-- 1--    {                           --beginning of the preamble--
--  --                                --font 0 is 10-point 'LPTA'--
-- 2--    [ xerox, xc82-0-0, lpta ] FINDFONT 100 SCALE MODIFYFONT 0 FSET
--  --                                --compute landscape page-to-Interpress transformation--
-- 3--    0.000035278 SCALE 0 -0.2159 TRANSLATE CONCAT 90 ROTATE CONCAT 1 FSET
--  --                                --compute portrait page-to-Interpress transformation--
-- 4--    0.000035278 SCALE 2 FSET
-- 5--    }                           --end of the preamble--
-- 6--    {                           --beginning of the first page body--
-- 7--    0 SETFONT                   --set the current font--
-- 8--    4 IGET 3 FSET               --obtain current transformation and save it in frame--
-- 9--    2 FGET CONCATT              --set the portrait page coordinate system--
--10--    720 7560 SETXY              --heading at x=1 inch, y=10.5 inch--
--11--    <Listing of GP0.PAS at 14:32 on 31 January 1982  Page 1> SHOW
--12--    3 FGET 4 ISET               --restore current transformation (see line 8)--
--13--    1 FGET CONCATT              --set the landscape page coordinate system--
--14--    720 5760 SETXY              --top line of listing at x=1 inch, y=8 inch--
--15--    <1  (* GP.PAS -- Simple PASCAL graphics package. *)> SHOW
--16--    720 5640 SETXY              --next line is 12 points below first line--
--17--    <2  const EnterGraphicsMode=29; LeaveGraphicsMode=31;> SHOW
--18--    720 5520 SETXY              --each line is 12 points below previous--
--19--    <3  var xlast,ylast: integer; v: InquiryResponse;> SHOW
--  --    --more lines of text for the first page would be added here--
--20--    }                           --end of the first page body--
--  --    --more page bodies for more pages would be added here--
--21--    END                         --end of the master--
```

The portrait-oriented coordinate system is set on line 9, using a scaling transformation set up on line 4. The landscape-oriented coordinate system is set on line 13, by concatenating onto

the current transformation a combined transformation set up on line 3. Note that the initial value of the current transformation is saved on line 8. This transformation, which establishes the Interpress coordinate system, is restored on line 12. This is necessary because the transformation that establishes the landscape page coordinate system on line 13 is defined relative to the ICS.

Although text is printed using two different orientations in this example, only a single font is prepared (line 2). The different orientations are achieved by using two different page coordinate systems. The call to SHOW on line 11 will print text horizontally because the current transformation in effect on line 11 has its x axis aligned horizontally, the transformations applied by SHOW leave the orientation of the axes unchanged, and the character operators in the font have the x axis aligned with the baseline of the character; hence the text will be horizontal. The other calls to SHOW, on lines 15, 17, and 19, print text rotated to run up the page. The reason is that the current transformation in effect for these calls has its x axis running up the page. Again, the transformations applied by SHOW leave the orientation of the axes unchanged, so the character operators will print characters with their x axis running up the page.

Multiple page coordinate systems are useful in many other situations besides the one illustrated above. A two-column format, for example, may be achieved by using one page coordinate system for one column and a second for the second column.

6.4 Summary

This section has presented the main features of the coordinate transformation machinery in Interpress. We have shown how transformations are used to:

- Establish a page coordinate system that has an orientation and unit of measurement that are convenient for the page being printed. Moreover, by manipulating the current transformation, the master can use several different page coordinate systems on a single page.
- Establish the size of characters. Although every character is defined in a standard character coordinate system in an Interpress printer's library, many different sizes can be obtained by specifying an appropriate transformation to MODIFYFONT.

These are the principal uses of transformations in Interpress. When transformations are studied in more detail in Section 13, we shall discover additional uses for transformations. We should also at this point update Figure 2.1 into Figure 6.8, to include the current transformation T.

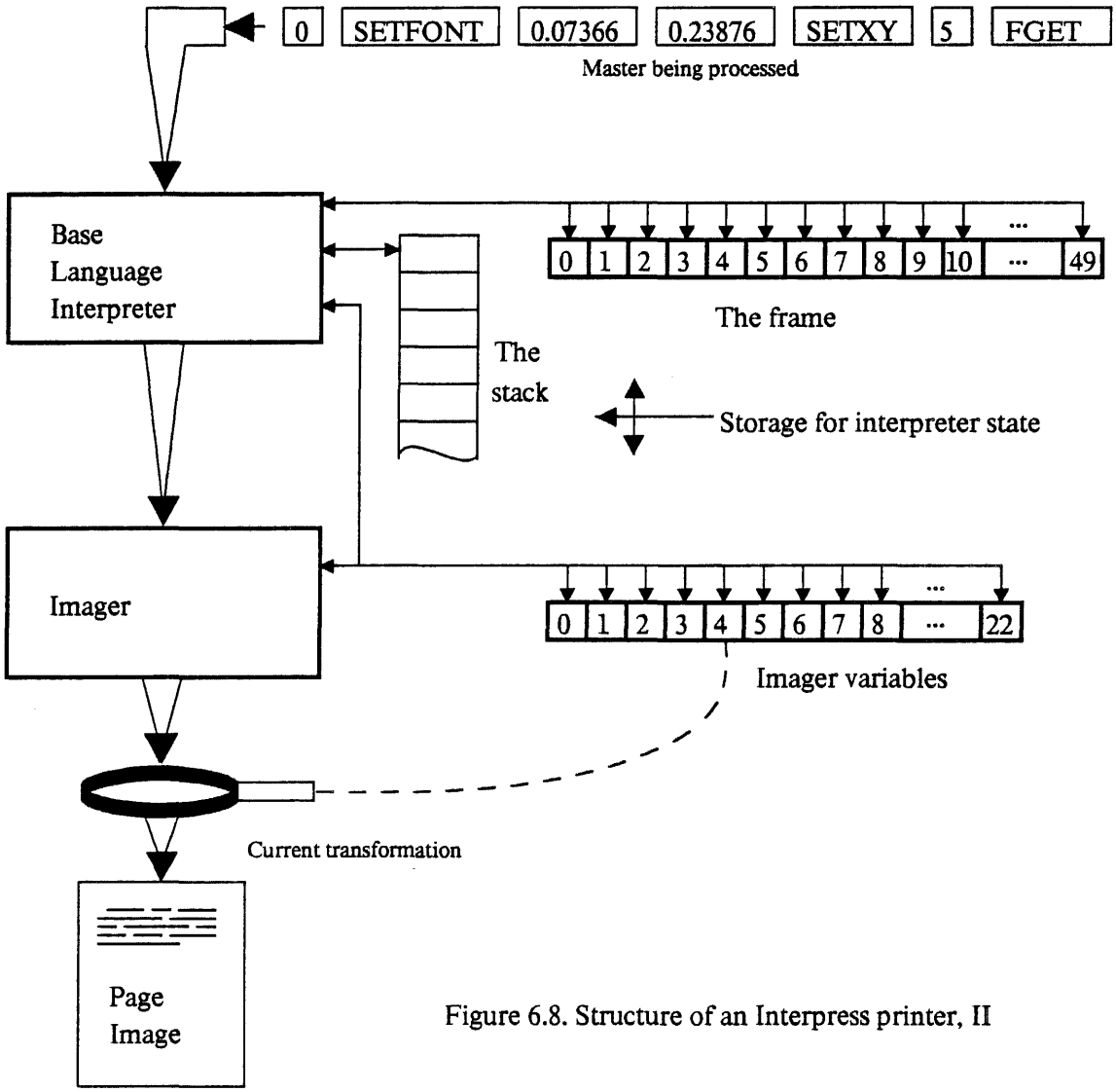


Figure 6.8. Structure of an Interpress printer, II

7

Creating masters: procedural interfaces

An application program that creates Interpress masters uses some form of *procedural interface*, a set of procedures called by the application program to generate the encoding. This section provides some suggestions about the design of that procedural interface. The approach suggested in this section is not part of the Interpress standard—how the application program is written is of no concern to the printer.

There are two kinds of procedural interfaces that you can build:

1. *Literal* interfaces, which provide a procedure for each kind of literal (or token) that can appear in the master.
2. *Operator* interfaces, which provide a procedure for each primitive operator.

This section is intended to be read by programmers who are designing programs that will make Interpress masters. Moreover, the reader should understand the Xerox encoding of Interpress, described in § 2.5, before reading this section further. This section makes numerous references to § 2.5.

7.1 Literal interface

A literal interface provides a procedure that corresponds to each Interpress type that has literals. The procedure appends to the encoding a literal of the corresponding type. The application program creates a master by calling these procedures to append each literal to the encoded master in the same order as the literals appear in the written form. (There is one exception to the ordering conventions, explained below in Section 7.1.1.)

The following set of procedures is sufficient for encoding all the masters in this Introduction:

```
InitializeMaster;                -- open an output file and put out the header --
FinalizeMaster;                  -- close the output file --

AppendOp(n: Integer);            -- output a primitive operator or symbol --
AppendNumber(r: real);           -- output a Number --
AppendInteger(n: Integer);       -- output an Integer --
AppendIdentifier(s: string);     -- output an Identifier --
AppendString(s: string);         -- used to encode <string> --
```

Before discussing the implementation of these procedures, we shall present an example of an encoding using this literal interface. Consider Example 3.2, which is reprinted here for reference:

```
--Example 3.2. A simple master to be generated by the literal interface--
--0--     BEGIN { }
--1--     {
--2--     [ xerox, xc82-0-0, times ] FINDFONT 0.00635 SCALE MODIFYFONT 0 FSET
--3--     0 SETFONT
--4--     0.07366 0.23876 SETXY
--5--     <Interpress> SHOW
--6--     }
--7--     END
```

The following Pascal-like program uses the literal interface to generate the master. A **const** declaration makes the arguments to *AppendOp* mnemonic rather than numeric.

const MAKEVEC = 283; FINDFONT = 147; SCALE = 164; MODIFYFONT = 148;
FSET = 21; SETFONT = 151; SETXY = 10; SHOW = 22;
BEGINMASTER = 102; ENDMASTER = 103; BEGINBODY = 106; ENDBODY = 107;

InitializeMaster; -- put out master header --
AppendOp (BEGINMASTER);
AppendOp (BEGINBODY); *AppendOp* (ENDBODY); -- null preamble --
AppendOp (BEGINBODY); -- begin page body --
-- set up font --
AppendIdentifier ("xerox"); *AppendIdentifier* ("xc82-0-0"); *AppendIdentifier* ("times");
AppendNumber (3); *AppendOp* (MAKEVEC);
AppendOp (FINDFONT);
AppendNumber (0.00635); *AppendOp* (SCALE);
AppendOp (MODIFYFONT);
AppendNumber (0); *AppendOp* (FSET);
-- now set current font --
AppendNumber (0); *AppendOp* (SETFONT);
-- set current position --
AppendNumber (0.07366); *AppendNumber* (0.23876); *AppendOp* (SETXY);
-- put out some text --
AppendString ("Interpress"); *AppendOp* (SHOW);
AppendOp (ENDBODY); -- end page body --
AppendOp (ENDMASTER); -- end of master --
FinalizeMaster;

7.1.1 Warning about body literals

In all of the examples we have seen so far, the order of literals in the encoded master is the same as the order of literals in the written form. *However, there is a set of cases in which this is not true.* Any operator that takes a body as an argument, a so-called "body operator," has a different encoding rule: the primitive operator literal encoding is placed immediately before the encoding of the body that is the argument. For example, the primitive operator CORRECT takes a single body as argument (§ 4.10):

⟨*b:* Body⟩ CORRECT → ⟨⟩

If the written form:

```
{ <Interpress> SHOW } CORRECT
```

were to be encoded, the encoding for CORRECT would come first, followed by the begin-body token, followed by literals in the body, followed by the end-body token. Thus the following calls would be made using the literal interface:

AppendOp(CORRECT);
AppendOp(BEGINBODY); -- begin argument body --
AppendString("Interpress"); *AppendOp*(SHOW);
AppendOp(ENDBODY); -- end argument body --

Of course, the literal interface could buffer the encoding of a body that is not part of the skeleton and wait for the trailing body operator before writing out the encoding. When the operator is given, the interface would write it to the encoding, followed by the literals in the body that have been buffered. Note, however, that this process must be recursive, i.e., it is possible to have body operators inside the body that is an argument to a body operator.

7.1.2 Implementing the literal interface

The implementation of the literal interface procedures depends on a detailed understanding of the encoding, given in § 2.5. The *AppendOp* and *AppendInteger* procedures are defined in that section. The discussion below supplements the discussion in that section.

Number. The procedure to encode a Number must choose an appropriate encoding mechanism, depending on the value of the number being encoded:

```
procedure AppendNumber(r: real);
    var d: Integer;
    begin
    -- If r is integral, use AppendInteger procedure, defined in § 2.5.2 --
    if r=trunc(r) and r≤maxInteger then AppendInteger(trunc(r)) else
        begin
        -- Must use a rational encoding. --
        -- Choose a value for d. See discussion below. --
        d := trunc(min(max(1,1000000/r),1000000));
        -- AppendRational is defined in § 2.5.2 --
        AppendRational(round(r*d),d)
        end
    end
```

To encode a non-integral number as a rational, a suitable denominator must be chosen. The procedure shown above chooses a denominator so that the rational approximation will be accurate to within one part in a million and so that the largest number used in the rational approximation is 1000000, unless of course *r* is greater than 1000000. The number 1000000 is chosen somewhat arbitrarily; another would do.

The best way to choose a denominator is to use information about the format of floating-point arithmetic in the creator's computer. A *real* is usually implemented in a computer as a rational approximation in which the denominator is a power of 2, so that a *real* can be encoded exactly

by using a denominator that is a power of 2. For example, suppose that a floating-point number is represented by two quantities e and i, such that the number is $r=i\times 2^e$, where i is an integer in the range $-2^{24}\leq i\leq -2^{23}-1$ or $2^{23}\leq i\leq 2^{24}-1$ or is exactly 0 and e is an integer in the range $-100\leq e\leq 100$. By rewriting the number in the form $r=i/(2^{-e})$, we find that a choice for the denominator is $d=2^{-e}$, unless $e>0$, in which case we'll have to choose $d=1$. Of course, we compute the numerator as $n=rd$. All that is required for this scheme is to be able to extract the value of e from a floating-point number.

If the details of the format of floating-point numbers are not known, we can indirectly compute the value that the denominator d must have. The following code will do just that on most computers, assuming *maxVal* is the maximum value of the denominator you wish to use in a rational approximation:

```
d := 1;
while (abs(r*d) < maxVal) and (d < maxVal) and (r*d ≠ trunc(r*d)) do d := d*2;
```

Primitive Operator. A procedure *AppendOp*(*n:* integer) is given in § 2.5.2 that will encode a primitive operator. Your program will be more readable if you use identifiers to stand for operators, as in the example above. Consult § B.3 for mnemonic codes for encoding primitive operators.

In addition to primitive operators, there are five special non-primitives that are encoded as Ops: BEGIN, END, PAGEINSTRUCTIONS, {, and }. These too may be given mnemonic identifiers, as in the example above.

Identifier. The following procedure appends an identifier to the master being created:

```
procedure AppendIdentifier(s: string);
    var i: integer;
    begin
    AppendSequenceDescriptor(sequenceIdentifier, s.length);
    for i := 1 to s.length do AppendByte(s.characters[i])
    end
```

This procedure assumes a programming-language type *string*, which Pascal does not have. It's obvious what the procedure is intended to do, however.

Body. It would be awkward to design a procedure to encode a body, because we would have to design a way to represent an arbitrary body in a data structure of our programming language so that it could be passed as an argument to the procedure. Instead, the application program can call *AppendOp* (BEGINBODY), then call various procedures in the literal interface in order to encode each literal in the body, and finally call *AppendOp* (ENDBODY).

Strings and Vectors. It is helpful to have procedures for converting vectors and strings in the programming language into the compact Interpress encodings for vectors and strings. While these procedures are not strictly necessary, since calls on MAKEVEC would do the job, the compact encodings they achieve are attractive. For example, we might use the following two procedures, described in § 2.5.3:

```
procedure AppendString(s: string);
procedure AppendIntegerVector(v: vector; numElements: integer);
```

Error checking. It is wise to outfit the procedures in the literal interface with some simple error-checking measures. In particular, *AppendOp* can check to be sure that Bodies are properly nested simply by counting the number of { and } symbols: the master is in error if the number of } symbols ever exceeds the number of { symbols or if the counts of the two symbols are not equal when the END symbol is output. Similar checking could be applied to BEGIN and END; in fact, it might be prudent to have *InitializeMaster* output the BEGIN and *FinalizeMaster* output the END, simply to avoid mistakes.

Summary. The virtue of the literal interface is that it can be used to encode an arbitrary Interpress master. However, it has some disadvantages:

- The program that generates the master is hard to read.
- It is easy to make trivial mistakes in the program, such as providing the wrong number of arguments to an operator.

These properties should be evident from the example above. The operator interface, described in the next section, remedies these problems.

7.2 Operator interface

An *operator interface* defines a separate encoding procedure for each important Interpress primitive operator. The encoding procedure takes the same arguments as the Interpress primitive operator. This interface is somewhat easier to use than a literal interface, because the application program is easier to read and because trivial mistakes are easier to avoid.

The easiest way to build an operator interface is on top of a literal interface. That is, procedures in the operator interface call procedures in the literal interface. For example, we might define:

```
procedure SetXY(x,y: real);
    begin
    AppendNumber(x); AppendNumber(y); AppendOp(SETXY)
    end;

procedure DrawLine(x1,y1,x2,y2: real);
    begin
    AppendNumber(x1); AppendNumber(y1);
    AppendNumber(x2); AppendNumber(y2);
    AppendOp(MASKVECTOR)
    end;
```

Using the first procedure, line 4 in Example 3.2 could be encoded with the call *SetXY*(0.07366, 0.23876). The application programmer will find this much more natural than the corresponding calls to the literal interface.

For primitive operators that take computed arguments or that return results on the stack, operator interface procedures can be built that omit computed results. Consider, for example, FGET: although it leaves a result on the stack, it takes an argument that is usually not a computed result. So it would be nice to have a procedure that handles the (usually) constant argument, just to avoid the tedium of using the literal interface. The procedures below show some examples:

procedure *FGet*(*n:* integer);
    begin *AppendInteger*(*n*); *AppendOp*(FGET) **end;**

procedure *FSet*(*n:* integer);
    begin *AppendInteger*(*n*); *AppendOp*(FSET) **end;**

procedure *Rotate*(*a:* real);
    begin *AppendNumber*(*a*); *AppendOp*(ROTATE) **end;**

procedure *Translate*(*x,y:* real);
    begin *AppendNumber*(*x*); *AppendNumber*(*y*); *AppendOp*(TRANSLATE) **end;**

procedure *Concat;*
    begin *AppendOp*(CONCAT) **end;**

procedure *ConcatT;*
    begin *AppendOp*(CONCATT) **end;**

Although the procedures for constructing transformations provide some help with the arguments to primitive transformations, the concatenation of transformations on the stack must still be dealt with much as in the literal interface. For example, line 3 of Example 6.6 could be encoded with:

Scale(0.000035278); *Translate*(0, −0.2159); *Concat; Rotate*(90); *Concat; FSet*(1);

The operator interface may also include procedures that handle frequently-used templates. Since the font setup template is used frequently, we might define a procedure that will generate the proper encoding:

procedure *SetupFont*(*fontNumber:* integer; *name:* string; *scale:* real);

This procedure could arrange to parse the font name, breaking it up into identifiers, e.g., the name "xerox/xc82-0-0/times" would be parsed into three identifiers and encoded.

7.2.1 An example using the operator interface

The same example used in Section 7.1 can be expressed with the following simpler program, which uses an operator interface:

```
BeginMaster; -- open file, put out master header and BEGIN --
 BeginPreamble;
 EndPreamble; -- null preamble --
 BeginPage; --start a page--
    SetupFont(0, "xerox/xc82-0-0/times", 0.00635); --font template--
    SetFont(0);
    SetXY(0.07366, 0.23876);
    Show("Interpress");
 EndPage; --end a page--
EndMaster; -- put out END and close file --
```

This example shows how separate procedures have been defined for *BeginMaster, EndMaster, BeginPreamble, EndPreamble, BeginPage,* and *EndPage.* This measure not only makes the program more readable but allows the interface to perform some error-checking, e.g., to check that *BeginPage* is not called inside a page body.

7.2.2 A suggested operator interface

While designing an operator interface is not difficult, it is often nice to have a place to start. Below is a list of suggested procedures and the literals they generate.

Operators with no arguments. Each primitive operator that takes no arguments has a procedure of the same name that simply outputs the appropriate primitive operator encoding, e.g., *Nop* will output ⟨NOP⟩. Operators of this type are: NOP, MOVE, TRANS, and STARTUNDERLINE. Procedures of this type offer no more power than the literal interface.

Operators with literal arguments. Operators that are used frequently with literal arguments are those that fit the operator interface best. Below is a list of possible procedures and the sequences they generate.

FGet(n)	⟨*n* FGET⟩
Mark(n)	⟨*n* MARK⟩
UnMark(n)	**if** *n*=0 **then** ⟨UNMARK0⟩ **else** ⟨*n* UNMARK⟩
IGet(n)	⟨*n* IGET⟩
Translate(x, y)	⟨*x y* TRANSLATE⟩
Rotate(a)	⟨*a* ROTATE⟩
Scale(s)	⟨*s* SCALE⟩
Scale2(sx, sy)	⟨*sx sy* SCALE2⟩
SetXY(x, y)	⟨*x y* SETXY⟩
SetXYRel(x, y)	⟨*x y* SETXYREL⟩
SetXRel(x)	⟨*x* SETXREL⟩
SetYRel(x)	⟨*x* SETYREL⟩
MakeGray(f)	⟨*f* MAKEGRAY⟩
SetGray(f)	⟨*f* SETGRAY⟩
MoveTo(x, y)	⟨*x y* MOVETO⟩
MaskVector(x1, y1, x2, y2)	⟨*x1 y1 x2 y2* MASKVECTOR⟩
MaskRectangle(x, y, w, h)	⟨*x y w h* MASKRECTANGLE⟩
MaskUnderline(dy, h)	⟨*dy h* MASKUNDERLINE⟩
MaskTrapezoidX(x1, y1, x2, x3, y3, x4)	⟨*x1 y1 x2 x3 y3 x4* MASKTRAPEZOIDX⟩
MaskTrapezoidY(x1, y1, y2, x3, y3, y4)	⟨*x1 y1 y2 x3 y3 y4* MASKTRAPEZOIDY⟩
SetFont(n)	⟨*n* SETFONT⟩
SetCorrectMeasure(x, y)	⟨*x y* SETCORRECTMEASURE⟩
SetCorrectTolerance(x, y)	⟨*x y* SETCORRECTTOLERANCE⟩
Space(x)	⟨*x* SPACE⟩

Operators with some arguments computed in the master and some literal arguments. These procedures take some literal arguments, but assume that other calls to the procedural or literal interface have invoked primitives to compute the other arguments. Below is a list of one possible set of procedures:

Get(n)	⟨*n* GET⟩
MakeVecLU(l, u)	⟨*l u* MAKEVECLU⟩
MakeVec(n)	⟨*n* MAKEVEC⟩
GetProp(id)	⟨*id* GETPROP⟩
FSet(n)	⟨*n* FSET⟩
Copy(n)	⟨*n* COPY⟩
Roll(depth, moveFirst)	⟨*depth moveFirst* ROLL⟩
ISet(n)	⟨*n* ISET⟩
MakeSampledBlack(transparent)	⟨*transparent* MAKESAMPLEDBLACK⟩
LineTo(x, y)	⟨*x y* LINETO⟩
LineToX(x)	⟨*x* LINETOX⟩
LineToY(y)	⟨*y* LINETOY⟩
MakeOutline(n)	⟨*n* MAKEOUTLINE⟩

Operator interface procedures in this class are often quite easy to use. For example, the trajectory procedures *LineTo, LineToX,* and *LineToY* can be used with *MoveTo* and *MakeOutline* or *MaskStroke. A call to MoveTo* starts a trajectory, calls to *LineTo, LineToX,* and *LineToY* continue the trajectory, and a call to *MakeOutline* or *MaskStroke* uses the trajectory in some way.

Operators with all arguments computed in the master. For those primitive operators all of whose arguments are typically computed, the operator interface includes a procedure of the same name that outputs the appropriate primitive operator encoding, e.g., *Exch* will output ⟨EXCH⟩. These operators are: SHAPE, POP, DUP, EXCH, COUNT, DO, DOSAVE, DOSAVEALL, CONCAT, CONCATT, GETCP, MASKFILL, MASKSTROKE, MASKPIXEL, MODIFYFONT, and all the test and arithmetic operators. Procedures of this type offer no more power than the literal interface.

Body operators. The following operators are body operators (Section 7.1.1): CORRECT, MAKESIMPLECO, IF, IFELSE, IFCOPY. Since literals for these operators must precede the body that is their argument, corresponding procedures in the operator interface take no arguments. However, it might be helpful to have *Begin- End-* pairs for each operator to improve error-checking, e.g., *BeginCorrect, EndCorrect.*

Templates. The following list shows some templates that might be useful to provide in the operator interface. The list is by no means complete; you may find it helpful to add new templates as the need arises. The notation [*name*] stands for a vector of identifiers that results from parsing a hierarchical name.

BeginPreamble, BeginPage, BeginBody	{
EndPreamble, EndPage, EndBody	}
Show(s)	⟨⟨*s*⟩ SHOW⟩
Fill(n)	*--note n trajectories must be on stack--* ⟨*n* MAKEOUTLINE MASKFILL⟩
MakePixelArray(xPixels, yPixels, vector, decompressionName)	*--note transformation must be on stack--* ⟨*xPixels yPixels* 1 1 1 6 1 ROLL *vector* [*decompressionName*] FINDDECOMPRESSOR DO MAKEPIXELARRAY⟩
FindColor(name)	⟨[*name*] FINDCOLOR⟩
SetStrokeWidth(w)	⟨*w* 15 ISET⟩
SetStrokeEnd(e)	⟨*e* 16 ISET⟩

SetupFont(fontNumber, name, scale)	‹[*name*] FINDFONT *scale* SCALE MODIFYFONT *fontNumber* FSET›
FindFont(name)	‹[*name*] FINDFONT›

7.3 Recommendation

It is usually worthwhile to build both literal and operator interfaces to encode Interpress masters. Since the operator interface is best built on top of the literal interface, the facilities of both can be made available to the application programmer with no extra effort.

An operator interface is appealing, because the application programmer can think of generating the image directly. The flow of information when generating and printing a master is:

application→operator calls→tokens→encoded master→tokens→imaging primitive calls→image

However, if the application program were using a *graphics package* to create an image on a device connected to the application computer, the flow of information would be:

application→operator calls→imaging primitive calls→image

This is a pleasing way to think about the process. The application program is written the same way regardless of whether an image is being made directly or an Interpress master is being prepared.

8

Software tools for Interpress

If you set out to write computer programs that generate Interpress masters, it's helpful to have some software tools to help track down troubles in the masters. In this section, we suggest some useful tools that can be built without much effort.

8.1 Encoded-to-written converter

The most basic tool is a program that reads an encoded Interpress master and prints out a written form of that master. It is a form of file-dumping program. This tool is of enormous importance in tracking down problems in masters. If a master fails to print on an Interpress printer, a "print out" of the master in a readable form is likely to reveal the problem. The encoded-to-written converter is also very useful when an Interpress master is obtained from some other organization and fails to print on your printer. The written form can reveal what fonts are required and other demands that the master makes of the printer.

The details of the presentation of the written form are less important than the legibility of the result. The written form used in the examples in this document and in the Standard is recommended.

The description of the written form in Section 2.2.1 overlooks some details that are important in converting all of the information in a master into written form:

- The notation **text** should be used to print the text enclosed in a *sequenceComment* token. The --text-- notation should be reserved for comments in the written form only.
- The notation ++text++ should be used to print the text enclosed in a *sequenceInsertFile* token.
- The first character of each identifier should be converted to lower case to avoid confusion between identifiers and primitive operator names, which are written in upper case.
- If the <string> shorthand notation is used, an escape convention will be required in order to print the character ">" if it appears in a string, or any non-printing character code. The recommended convention is to print *#number#* for an element of the string whose value is *number*. Thus, for example, ">" would appear as #62# if the character set being used is ISO 646. Of course, because # is used as an escape character, it too must be printed numerically.

It is important that this tool be able to produce readable output for any master, regardless of how ill-formed it is. It is therefore crucial that the encoded-to-written converter never abandon the decoding process because of what it thinks are errors. This tool should do nothing more than print the contents of a master in a readable format. Another diagnostic tool, the check interpreter described in Section 8.3, can be used to find errors in the master.

It is probably worthwhile to add a few conveniences to the encoded-to-written converter. The program might allow skipping selected page bodies as it converts to the written form, so that if there is a problem with some particular page of a master, only that page needs to be listed.

8.2 Written-to-encoded converter

A second tool that converts from the written form to the encoded form is also helpful. It serves two principal functions. First, it allows you to type into a text editor an arbitrary Interpress master in order to try out some of the examples in this document or to experiment with various Interpress features. Second, if you use it in conjunction with the encoded-to-written converter, you can repair bugs in masters by converting them into the written form, altering the text with a text editor, and then converting back to the encoded form. This feature is helpful when debugging new creation software and when dealing with an odd-ball master that may have originated outside your environment.

An important property of the two converter programs is that they be exact inverses of one another. If an encoded master is converted to written form and then (without changes) converted back to encoded form, the result should be identical to the original master. This property is essential to avoid loss of numeric precision in the conversions and to ensure that when the written form is edited, the encoded master obtained by converting it will differ from the original only due to the edit.

The written form may have to be designed carefully to meet the somewhat conflicting objectives of legibility and invertibility. For example, a rational number encoded as *n/d* must be represented in the written form as *n/d* to insure that a conversion back to the encoded form will use the same denominator as the original master. However, the decimal fraction form of this number should probably appear as a comment immediately following the rational form so that the written form can be read easily, e.g., 17/11 --1.545--.

The written-to-encoded tool must be careful to treat body operators properly (see Section 7.1.1). In the written form used in this document, a body argument precedes its body operator, but in the encoded form, a body operator precedes its body argument. If a similar written format is chosen, both converter tools will need to reverse the order of body argument and body operator. Alternatively, you might choose a written form in which literals appear in precisely the same order as in the encoded form.

8.3 A check interpreter

If you're doing a lot of Interpress programming, it may be worthwhile to build a *check interpreter*, a program that interprets an Interpress master and scrupulously checks all the rules of the Interpress language. It should check that all arguments to operators are of the correct type. It should also verify that no *limits* are exceeded; in fact, it might keep track of the largest amount used of all the limited resources (e.g., largest body, highest frame index used, largest vector, largest Integer, and so on).

The check interpreter could be an Interpress printer with comprehensive error-detection and diagnostic features in the interpreter software. The output of this printer could be the diagnostic information in addition to (or instead of, if the master is full of errors) the regular printed output. Alternatively, the check interpreter could be an interactive program that lets you con-

trol the execution of the master, setting breakpoints and single stepping if needed, and letting you probe the contents of the stack, the frame, and the imager variables as needed. An interactive check interpreter could also show the page image on a display, adding each new component of the image as it is produced.

8.4 A graphics package

Another piece of diagnostic software that helps check out creator software is a graphics package that has the same interface as the operator interface described in Section 7.2. The graphics package, rather than building an Interpress master, will create a display of the page as it would have been printed. Moreover, the display can be created as each operator-interface procedure is called, so that the programmer can debug his or her application program interactively.

It is essential that the graphics package have the same interface as the operator interface. A programmer who uses the graphics package for debugging can then be assured that the program will work the same way when actually creating a master. Of course, it would be possible to combine the graphics package and the operator interface to Interpress, so that a run-time switch would determine whether a master or a display was to be generated.

A graphics package that mimics the operator interface for the operators we have introduced so far is described by Warnock and Wyatt [24]. Operators that take *bodies* as arguments, however, will cause problems, because the graphics package will have to save the contents of such bodies. The most frequently used operator of this sort is CORRECT, described in Section 10 and § 4.10.

8.5 Recommendation

The two converter programs are simple to program and extremely valuable. The graphics package and check interpreter, on the other hand, are major undertakings, justifiable only if you're doing a lot of Interpress work. Perhaps someone will offer a checking service: you send an Interpress master via computer network to the service and receive in return a diagnostic listing of the check interpreter's results.

9

Fonts

Although Interpress can be used to specify almost any image at all on a printed page, images containing letters, numbers, and other text characters are common enough that Interpress has a special mechanism for dealing with them. The mechanism is patterned after the mechanism of movable type that has been in use for centuries, and much of the vocabulary necessary to understand it and use it comes from typesetting.

To draw a character on a page, we need to add to the page an image of the character, drawn in a certain way, in a certain size and rotation. A character's shape is called a *letterform*, and placing an instance of a character on a page is called *imaging a letterform*. Most of the earliest models of phototypesetting machines work by having actual mechanical letterforms, photographic negatives, that are imaged by shining a bright light through them onto photographic paper, one letter at a time. Interpress does not use photographic negatives or mechanical letterforms, of course, but you can think about the mechanisms in the same way.

Interpress accommodates letterform definitions of many different sorts, and can take advantage of the high-quality typefaces designed for photocomposition and printing.

A *font* is a collection of letterforms designed to be used together. When a master is prepared, the creator needs to know a good deal about these fonts in order to format the document properly. To decide how many characters will fit on a line, the creator must know the "width" of each character, i.e., the amount by which the current position is moved when the character image is placed on the page. Likewise, to justify a line of text between margins, the creator must know character widths.

In order to allow a creator to predict the appearance of characters it places on the page, Interpress fonts adhere to rigid conventions governing:

- The precise actions that are taken when an individual character is placed on the page by SHOW.
- Reporting to the creator various *metric* information concerning the font and each character in it, such as the character's width.

However, Interpress' treatment of fonts is deliberately flexible in two important areas:

- The conventions used to *name* a font.
- The choice of a *character set* for a font, that is, the correspondence between character codes and shapes.

In the areas of character sets and font names, the Interpress standard is designed to accommodate a wide range of current practice in the computing and printing industries, rather than to impose new standards. Both of these areas are characterized by long and interesting histories that have led to diverse practices. Rather than choosing a particular practice and standardizing it within Interpress, Interpress allows arbitrary character sets to be used and provides a framework in which almost arbitrary font names can be used.

For a discussion of the historical and technical issues surrounding fonts, we recommend the Updike book [23]. For a representative sample of type faces, the International Typeface Corporation's font catalog [12] is superlative.

9.1 Font names

Interpress names a font with a vector of identifiers, for example [*itc, ascii, times*]. Names of arbitrary complexity are handled by extending the vector to contain any number of identifiers. The name may capture any or all aspects of a font, such as its style, the name of its designer, or a version number. Font names used in the printing industry can be mapped quite easily into this framework. Thus "Bodoni Bold Condensed" might become [*itc, ascii, bodoni, bold, condensed*], and similarly, Caslon Demi-bold might become [*itc, ascii, caslon, demibold*].

In the above examples, the identifiers *itc* and *ascii* precede the identifiers for what might conventionally be considered the font name. These identifiers are examples of *hierarchical naming conventions*, and are the only aspect of font names that Interpress insists on. Interpress insists on hierarchical names so that different Interpress users and applications can devise font names that are unique, that is, that do not conflict with anyone else's invented names. The way that hierarchical names are organized is described in Section 11. Once past the first few identifiers that satisfy hierarchical name requirements, however, you are free to organize font names in whatever fashion you wish. Naturally the font naming scheme must allow masters to refer to fonts by name and Interpress printers to find and use the fonts.

A font name does not contain information about the geometric properties of a font, such as its size and orientation. These are controlled not by the name but by the geometric transformations specified to MODIFYFONT and the current transformation in effect when SHOW is called. However, the highest quality typographic practice requires that letterforms of different sizes have different shapes—letterforms are not simply magnified or reduced. In these cases, the different type designs are given different names. For example, one common technique is to define three different letterform designs, one for very small type such as used in footnotes, one for common "body" text, and one for headline or "display" type. These properties could be part of the font name, as in [*itc, ascii, caslon, demibold, display*]. It's important to realize, however, that the optimum letterform design is related to the viewing angle it subtends at the eye, not to its physical size. Body characters on a billboard may be 50 cm. high, while the body characters on this page are only 0.35 cm. high. These two uses of body characters will use the same letterform design, but will have different geometric transformations applied.

9.1.1 Xerox font names

Fonts whose hierarchical names begin with [*xerox*, . . .] are subject to a naming standard being developed by Xerox. At the time of this writing, these standards are not complete. However, the example masters in this Introduction all use the following convention:

1. The first identifier is *Xerox*.
2. The second identifier specifies the character set used by the font (see Section 9.2). The Xerox Character Code and Rendering Code Standards [25, 29] are denoted by *xc82-0-0*.
3. The third identifier is the "typographical name" of the font, encoded in a single identifier. Examples are *times*, *timesitalic*, and *timesbold*.

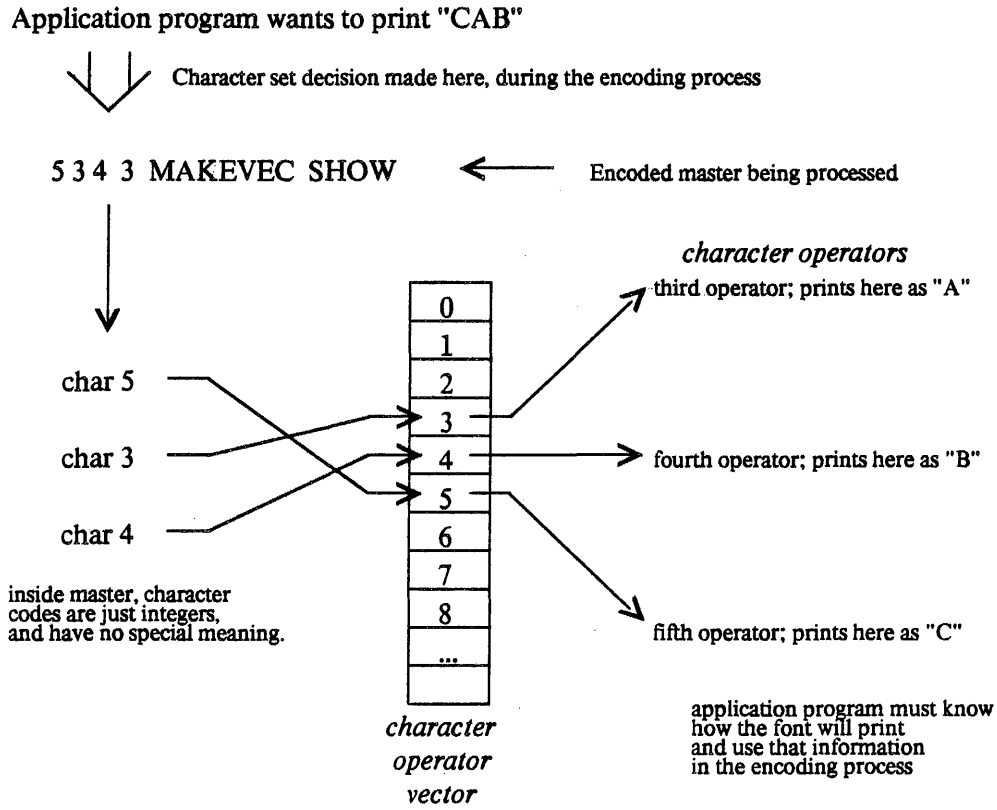


Figure 9.1. Steps in imaging a character on the page

9.2 Character sets

The term *character set* refers to the correspondence between a digital *character code* and its printed, or *graphic*, form. The notion of character set is clearly vital to Interpress, which must make graphic images on a page according to digitally-encoded instructions. Many different character sets are used today, and some have been standardized (e.g., ASCII, ISO 646, EBCDIC)—some of these standards are described in Section 23.3.1. But each of these standards is unacceptable for some applications, and none has become dominant. As a result, Interpress

is carefully designed not to depend in any way on the choice of character set for a font. The SHOW operator is given as an operand a vector of character codes, which it uses to index into a vector of character operators to find a *character operator*, which actually places an image on the page. Figure 9.1 shows this operation pictorially.

An Interpress printer treats all characters in this way, with no knowledge of what a character code signifies. It is the master that controls both the choice of character codes passed to SHOW and the choice of the vector of character operators to use; thus the character set is of no consequence to an Interpress interpreter and printer.

9.2.1 A character set analogy: daisy-wheel typewriters

An analogy may help to clarify the nature of a character set; please refer to Figures 9.2 and 9.3. Consider a typewriter that uses the "daisy wheel" printing technology. Let's ignore the labels printed on the typewriter keys, and instead print on each key a number from 1 to 96 that corresponds to a position on the wheel. The typewriter mechanism responds to a keystroke on key *n* by rotating the wheel to position *n* and striking the wheel sharply so as to imprint whatever character shape has been cast into the wheel at that position. The typewriter is thus completely insensitive to the identity of the characters in the images it makes. Instead, the user controls the character selection because he controls both the keys struck and the selection of the wheel to use. This situation corresponds exactly to the situation in Interpress; the identity of the character images is controlled entirely by the master and not by the Interpress interpreter or imager. The character codes in the operand passed to SHOW correspond to key numbers and the font established by SETFONT corresponds to a wheel selection.

The analogy helps illustrate another aspect of character sets as well. In the daisy-wheel typewriter, keys are labeled with characters (e.g., A, B, C) because most, but far from all, daisy wheels use the same character set. So a wheel for roman type and one for italic type will use the same character set—this avoids confusion and is of great convenience to the user because it allows keys to be labeled in a meaningful way. But some wheels contain many special symbols that do not correspond to the key labels. In this case, a plastic chart overlaid on the keyboard tells which key to strike in order to obtain each special character. This chart is somewhat like a character set, in that it defines the correspondence between character codes (typewriter keys) and printed symbols. However, it is not completely analogous to a character set, because it is a mapping between the special key meanings and the regular key meanings, rather than a mapping between the special key meanings and print wheel positions.

It is extremely important to understand that *Interpress imposes no standard character set* when printing images of characters. If the master selects a font designed to use the ASCII character set, calls to SHOW must have operands containing ASCII character codes. If the same master later selects a font designed to use the EBCDIC character set, subsequent calls to SHOW must provide EBCDIC character codes. Still other fonts may use no standardized character set, but simply choose character codes at will like the daisy-wheel for special characters. Fonts of arbitrary character set can be used within the same master just as daisy-wheels of arbitrary character set can be used to type on the same page.

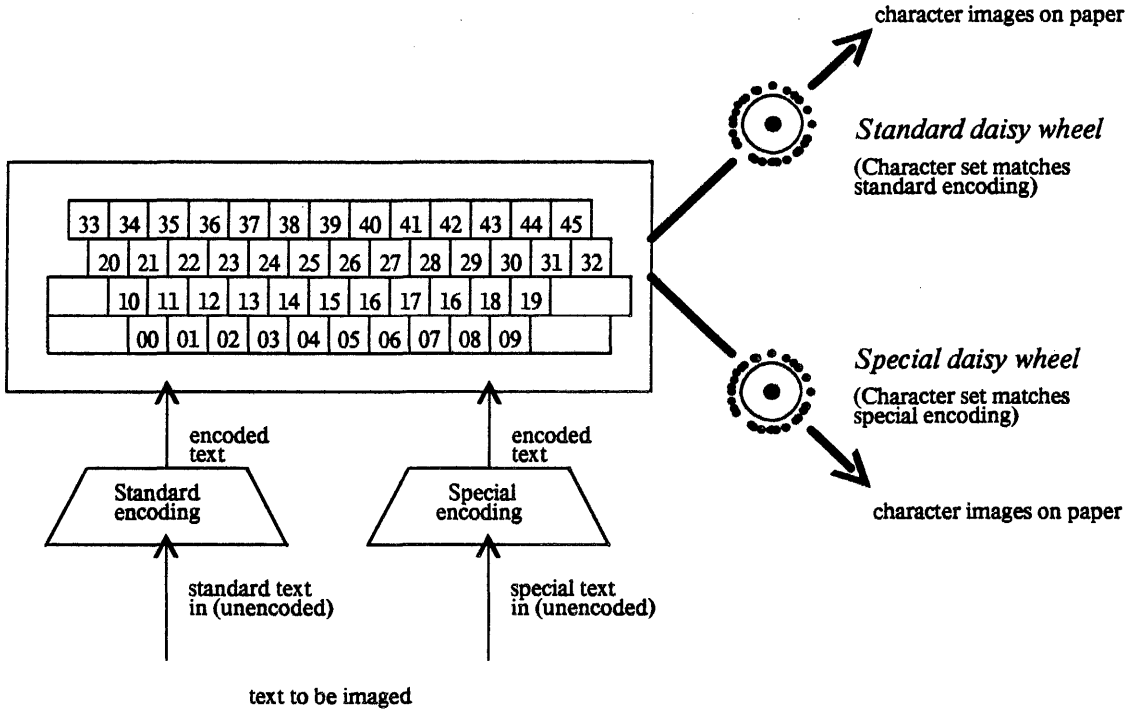


Figure 9.2. A daisy-wheel printer that assumes no character set

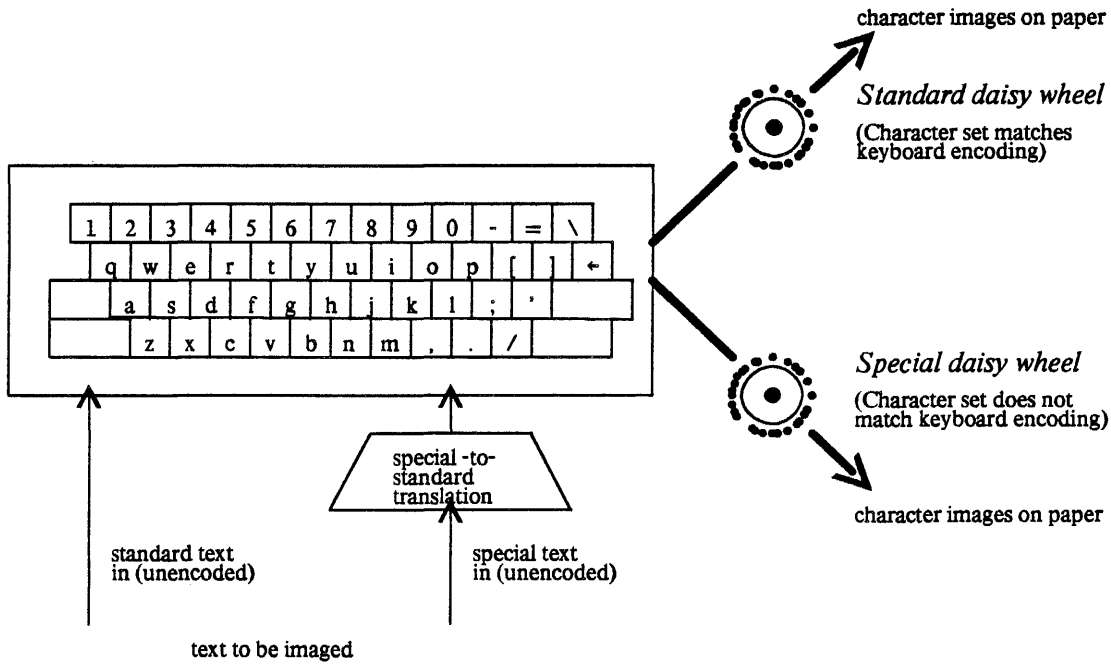


Figure 9.3. A daisy-wheel printer that assumes a character set

9.2.2 Character sets and the Interpress encoding

Although the Interpress language does not have a standard character set, the *encoding* of Interpress does make use of a standard character set in two cases. The encoding of identifiers and of the header uses the ISO 646 character set to obtain a numeric code for the characters and symbols that appear in these two objects. But this use of a standard character set has *nothing* to do with character sets of fonts or with making images of characters; it merely provides a standard way of interpreting font names and a readable transcription of the header.

9.2.3 Determining the character set of a font

The creator must know the character set of the fonts it uses. In most cases, a given creator is designed to work with only *one* character set and selects fonts that are known to observe the necessary character set conventions.

Font names can be devised so as to communicate font properties to the creator. In particular, a printer might use a convention that incorporates the name of the character set into the hierarchical font name. For example, Xerox's computer printing products might adopt the convention that one level of the name hierarchy is devoted to character set identification. Thus fonts with names of the form [*xerox, ascii, . . .*] use an ASCII character set, those of the form [*xerox, ebcdic, . . .*] use the EBCDIC character set, those of the form [*xerox, xc82-0-0, . . .*] Xerox Character Codes, and so on.

9.3 Character operators

Each character in a font is represented by a *character operator*, which is responsible for making an image of the character on the page whenever it is executed. A printer's font library will contain definitions for character operators for the characters of each font. As we shall see in Section 14, it is possible for a master to define character operators itself, useful for special characters that are not available in the font library.

A character operator must do three things:

1. Generate an image of the character. Interpress' graphical operators are used for this purpose. Recall that the character's shape is specified in the character coordinate system (Section 5.3.3), in which the unit of distance measurement is the "point size" or "body size" of the character and the x axis lies along the character's *baseline*. The placement, size, and orientation of a character placed on the page is controlled by the transformation applied to the geometry in the operator when it is called by SHOW.
2. Change the *current position* so as to prepare for imaging the next character in a sequence. This is accomplished by moving the current position so as to correspond to the point (*widthX, widthY*) of the character.
3. Adjust spacing a small amount. Small adjustments to character spacing may be required in order to compensate for small differences between character widths used in the printer and those assumed by the creator. This topic is covered in detail in § 4.9, § 4.10, and Section 10.4.3.

9.3.1 Graphic characters

The three activities that must be performed by a character operator are illustrated by a character operator for the character "+":

```
--Example 9.1: sample definition of a character operator--
--0--     {                    --the character operator is defined by the following body literal--
--1--     0.1 16 ISET          --set strokeWidth to 0.1 units--
--2--     0 16 ISET            --set strokeEnd to 0 (square) --
--3--     0.25 0.35 0.65 0.35 MASKVECTOR --horizontal bar of '+'--
--4--     0.45 0.15 0.45 0.55 MASKVECTOR --vertical bar of '+'--
--5--     0.9 0 SETXYREL       --set current position to (0.9,0)--
--6--     CORRECTMASK          --call an operator to correct spacing--
--7--     } MAKESIMPLECO       --the bracket that ends the body literal--
```

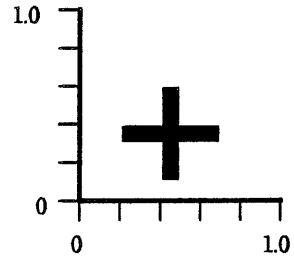


Figure 9.4. The character "+" in its coordinate system.

Figure 9.4 shows the character defined by this operator. Lines 0 and 7 are used to define a composed operator, an operation that will not be described until Section 12. It suffices for the moment to explain that the body (lines 1 through 6) will be executed whenever this character operator is invoked. The three duties of the character operator show clearly in the body:

1. Generating the image of the character. The first four lines (1–4) are concerned with generating the image of the "+". The example uses strokes for this purpose, as illustrated in Section 3.1, although most character operators will use more sophisticated graphics primitives in order to obtain more legible character shapes. The graphical primitives are described in Section 15 or § 4.8.2.

2. Moving the current position. Line 5 calls the operator SETXYREL to alter the current position. Of course, when the character operator is invoked, the current position is at the origin of the character coordinate system. Line 5 will change the current position to the point (0.9, 0) in the character coordinate system. This point has been determined by the font designer to be the place where the origin of the next character in a sequence should lie.

3. Adjusting the spacing. Line 6 calls the operator CORRECTMASK to provide an opportunity for spacing correction. This operator may change the current position by a small amount, as described later (also see § 4.10). If the creator and the printer are in exact agreement about font widths (e.g., no font approximations have been made), CORRECTMASK will not change the current position at all.

This example illustrates clearly what a character operator does. It also shows what the font designer must do: he must devise a set of calls to Interpress graphical operators in order to create an image of the character; and he must decide where the next character in a text sequence should be placed, i.e., he must determine the width of the character.

A printer's font library contains a great many character operator definitions like the one above. While they may be represented in some form besides that of an Interpress body for efficiency reasons, they achieve the same effect. The printer's font library is thus a convenience but not a necessity: a master could provide a definition for every character operator it uses, but the master would grow in size and the interpretation of the master by the printer would doubtless slow down.

9.3.2 Spaces

Some "space" characters have no graphic image, but have a non-zero width. These space characters fit into the character operator paradigm exactly—except that no graphic image is produced. The following example illustrates how a space character operator might be defined:

```
--Example 9.2: a character operator for the "space" character--
--0--     { --no graphics--
--1--     0.25 0 SETXYREL          --set current position to (0.25,0)--
--2--     0.25 0 CORRECTSPACE      --call an operator to correct spacing--
--3--     } MAKESIMPLECO
```

In this example, the character operator simply changes the current position in order to place the next character in sequence at the proper spot. Note that this space character invokes the CORRECTSPACE operator to perform spacing correction, while the operator in Example 9.1 invokes CORRECTMASK. The reasons for this distinction are explained later, in Section 10.4.3.

9.3.3 Amplifying characters

Interpress allows a character operator to use an alternate spacing computation, one that scales a character's width by the value of an imager variable named *amplifySpace*. Characters that use this kind of spacing calculation are termed *amplifying* characters. The most common use is to place in a font a single amplifying space character, which is used to achieve simple kinds of text justification, explained in Section 10. An example of a character operator for an amplifying space is:

```
--Example 9.3--
--0--     { --no graphics--
--1--     0.25 18 IGET MUL 0 SETXYREL        --change current position by (0.25*amplifySpace,0)--
--2--     0.25 18 IGET MUL 0 CORRECTSPACE    --call an operator to correct spacing--
--3--     } MAKESIMPLECO
```

Note that the operator invokes other operators to obtain the value of *amplifySpace* (imager variable 18) and to perform the multiplication. While this computation looks somewhat like a geometric transformation, it is best viewed instead as if *amplifySpace* were a global variable that acts like a parameter to the character operator.

9.3.4 What character operators cannot do

Restrictions on Interpress character operators prevent a one-to-one mapping between some existing character sets and Interpress characters. The principal limitation is that a character operator is allowed to change the current position in one of only two ways: the current position is advanced either by the character width or by the amplified character width.

There are two practices in some existing character sets that cannot be achieved with Interpress characters:

1. Formatting characters. Many character sets contain formatting characters other than spaces that are used to control the format of presentation of text. Examples from ASCII are carriage-return, tab, form-feed, and vertical-tab. An Interpress character operator cannot achieve the effect of these characters. The reason is that the current position cannot be reset to a particular value (carriage-return) or to one of a number of fixed values (tab, vertical-tab). These formatting operations are achieved in Interpress with positioning commands such as SETXY rather than by interpreting character codes.

2. "Automatic" ligatures. High-quality typography often makes use of *ligatures*, which are special graphic forms designed to show sequences of characters in a row. For example, two "f" characters in a row may appear as a ligature "ff" rather than as two separate characters "ff". You may have to look closely to see the difference in text, but it should be more visible in these enlarged characters:

Letter pairs: ff fi Ligatures: ff fi

In Interpress, each such ligature must have a separate character operator in the font; a ligature cannot be obtained "automatically" by showing the two characters in sequence. So the master, rather than invoking [102, 102] SHOW to obtain two separate "f" characters, might invoke [136] SHOW to obtain the ligature—the ligature is simply another character in the font. Automatic ligatures cannot be achieved by Interpress operators because of the restrictions on the width calculation: the width of a character cannot depend on the identity of the preceding character, as it would have to for automatic ligatures. This is not a serious restriction, because the creator is perfectly capable of recognizing character pairs as it outputs them and performing the appropriate substitution. Moreover, this strategy adheres to an important design goal of Interpress: all decisions about presentation and formatting should be made by the creator, not the printer.

9.4 Font metrics

Although a creator does not need to know all details of a character operator in order to create a master, a certain amount of *metric information* is necessary, e.g., the character's width and whether it uses the amplifying width calculation. Interpress defines a set of metric properties for each character and some metric properties that apply to an entire font.

The full set of character metrics is explained in § 4.9.3. The most important character metrics, which Interpress records about every character, are:

widthX, widthY: Number. These two numbers give the width of the character. If *widthX* is not specified, it is assumed to have value zero. If *widthY* is not specified, it is assumed to have value zero.

amplified: Integer. This metric tells whether the character is amplifying. If *amplified*=0, the calculation that changes the current position is ⟨*widthX widthY* SETXYREL⟩; if *amplified*=1, the calculation is ⟨*widthX amplifySpace* IGET MUL *widthY amplifySpace* IGET MUL SETXYREL⟩. If the *amplified* metric is not specified, it is assumed to have value zero.

Character metrics may also contain information about ligatures in the font. For example, in the character metric information for "f" there is a ligature table that says that if this character ("f") is followed by one specific character ("f"), a ligature for the two-character sequence is available as character code 136, or if followed by another specific character ("l"), another ligature is available.

All of the information about a font in a printer's library is captured in a Vector called a FontDescription. The elements are (§ 4.9.1):

operators: A Vector of composed operators: these are the character operators themselves. Examples 9.1 through 9.3 give an indication of what a character operator is like. The index of an operator in the *operators* Vector is called the *character index,* or sometimes the *character code.*

characterMetrics: A Vector of metric information for each character that appears in *operators.* The most important metrics are *widthX, widthY,* and *amplified.* The format of a CharacterMetrics vector is described in § 4.9.3.

metrics: A Vector that gives metric information for the entire font. The contents of this vector are optional, and are described in § 4.9.3.

name: A Vector of Identifiers. This vector gives the full name of the font. This name can be passed to FINDFONT to extract this font from the printer's library.

Most of this information is not directly accessible to a master as it is being interpreted. The *operators* vector is obtained by FINDFONT, but no other elements of a FontDescription can be examined within the master. However, the other information in a FontDescription is made available to creators by a mechanism described in the next section.

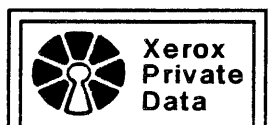
9.5 Communicating metrics to the creator

In order to prepare a master, a creator must be able to obtain metric information for the fonts it will use in the master. This is accomplished by providing the creator with metric information for all the fonts that a printer has in its library. For each font, the creator is provided with the FontDescription vector described in the preceding section, which includes font-wide metrics as well as individual character metrics (the *characterMetrics* vector). Most important, this vector contains the *name* of the font, which can be used in a call to FINDFONT to obtain the font.

Metric information is encoded using *property vectors,* which are designed so that certain metric properties can be given and others can be omitted. A property vector is a Vector constructed using a convention that elements are paired together: the first element in each pair is a *property name,* usually an Identifier; and the second element in the pair is a value corresponding to the named property. For example, the property vector [*widthX,* 21, *widthY,* 14] records that the *widthX* property has the value 21 and the *widthY* property value 14.

Font metric information, represented as property vectors, is presented to the creator in the form of a *metric master.* This is an Interpress master that has a preamble but no page bodies. The execution of the preamble of the metric master will leave on the stack one or more FontDescription property vectors, one corresponding to each font in the printer's library. Usually, these vectors contain only the *characterMetrics, metrics,* and *name* properties. In order to simplify interpretation of the metric master, there are restrictions on the set of Interpress primitive operators it may contain (§ 4.9.3).

A portion of a metric master is shown below. It describes a font library containing two fonts, [*xerox, ascii, times*] and [*xerox, ascii, timesitalic*]. The first font has character indices from 32 to 126 inclusive; the second from 32 to 136. The second font shows how ligatures for the character "f" might be recorded. In order to understand this master thoroughly, you will need to read §§ 4.9.1–4.9.3. The master is presented in abridged form, for the sake of clarity.



```
--Example 9.4. A metric master --
-- 0--    BEGIN {                          --start of preamble--
-- --     --construct first "font" vector--
-- 1--      characterMetrics               --property name for "characterMetrics" vector--
-- 2--      --construct the "characterMetrics" vector--
-- --         --construct the character metric information for char 32--
-- 3--        32                           --property name is character index--
-- 4--        widthX 0.34 widthY 0 amplified 1    --widthX=0.34, widthY=0, amplified=1--
-- 5--           6 MAKEVEC                 --actually make the 6-element vector value for char 32--
-- --         --construct the character metric information for char 33--
-- 6--        33 widthX 0.24 2 MAKEVEC  --widthY=0, amplified=0 because omitted--
-- --         --construct the character metric information for char 34--
-- 7--        34 widthX 0.28 2 MAKEVEC
-- --         --. . . and so on, for characters 35 through 126--
-- 8--        190 MAKEVEC                  --make "characterMetrics" property vector--
-- --                                      --since each character uses 2 element, the total size--
-- --                                      --of the vector is 190=(126-32+1)*2--
-- 9--      name                           --property name for hierarchical name vector--
--10--        xerox ascii times 3 MAKEVEC  --make a "name" vector--
--11--    4 MAKEVEC                        --make FontDescription vector for [ xerox, ascii, times ]--
-- --
-- --     --construct second "font" vector--
--12--      characterMetrics               -- property name--
--13--        32 widthX 0.36 amplified 1 4 MAKEVEC     --character metrics for char 32--
--14--        33 widthX 0.26 2 MAKEVEC  --character metrics for char 33--
--15--        34 widthX 0.30 2 MAKEVEC  --character metrics for char 34--
-- --         --. . . more character metrics for chars 35 to 101--
--16--        102 widthX 0.30              --character code 102 ('f') to show ligature--
--17--          ligatures                  --property name for ligatures vector--
--18--            102 136 2 MAKEVEC        --'ff' ligature is code 136--
--19--            108 137 2 MAKEVEC        --'fl' ligature is code 135--
--20--          2 MAKEVEC                  --"ligatures" vector--
--21--        4 MAKEVEC                    --character metrics for char 102--
-- --         --. . . more character metrics for chars 103 to 135--
--22--        136 widthX 0.58 2 MAKEVEC --character metrics for char 136--
--23--        210 MAKEVEC                  --"characterMetrics"--
--24--      name                           --property name for hierarchical name vector--
--25--        xerox ascii timesitalic 3 MAKEVEC --"name"--
--26--    4 MAKEVEC                        --FontDescription for [ xerox, ascii, timesitalic ]--
--27--    } END                            --end of preamble and metric master--
```

It is common in typography to define fonts with widths expressed in *units of set*, equal to 1/18 em, i.e., 1/18 of the body size of the type. Sometimes units of set are defined as 1/54 of the body size. In either case, the metrics master can express these widths exactly, without roundoff error, using the *sequenceRational* encoding form, e.g., 13/54 or 11/18.

The metric master, unlike most Interpress masters, is used to convey information from the printer to the creator (Figure 9.5a). Interpress does not define how metric masters are created or how a creator obtains access to one. The idea is that each printer should be able to produce a metric master that describes its font library and to record it on transportable media or to transmit it to the computer where the creator executes. There it will presumably be stored as a file that the creator can access repeatedly (Figure 9.5b). The metric master is encoded using the same encoding rules used to make ordinary masters. While such an encoding is standard and easy to describe, it may not be particularly convenient. If it is deemed unsuitable for a creator to contain software capable of interpreting the Interpress operators in a metric master, then a simple conversion program can perform this evaluation once and write the font information into a file in a private format; the creator can then read this file to find its font metrics. This scheme is pictured in Figure 9.5c. Interpress does not define metric representations other than the metric master, nor does it prohibit them.

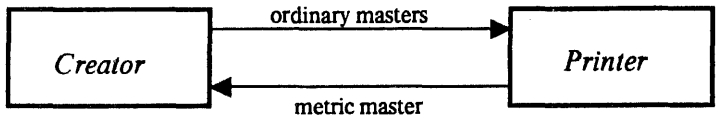


Figure 9.5a. Direct communication of metric masters.

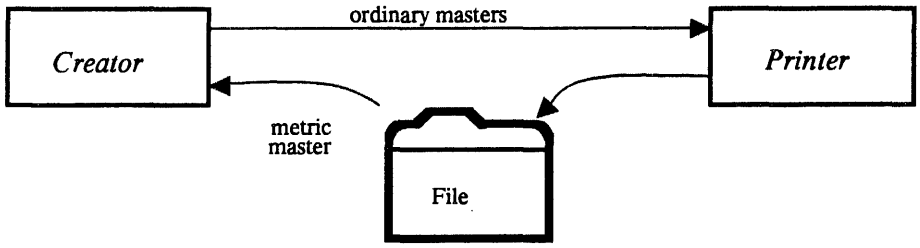


Figure 9.5b. Indirect communication of metric masters.

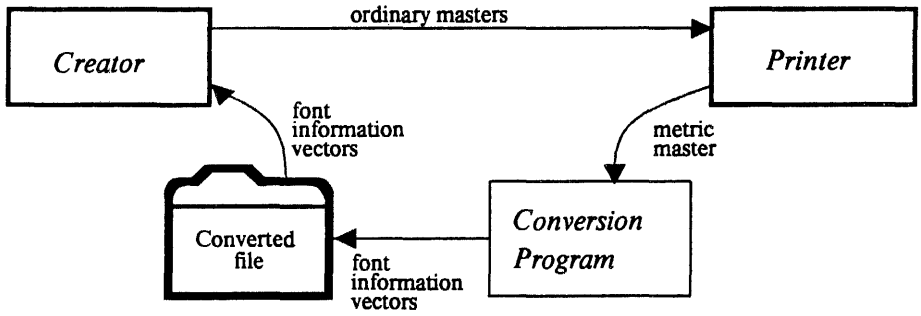


Figure 9.5c. Converting a metric master into another form

9.6 Font libraries and printer font storage

It is impractical for every printer to contain every font, but it is overly restrictive to require a creator to limit his font usage to ensure that every printer will have a reasonable chance of being able to print a document. This section describes a few conventions for font libraries and for action to be taken in case a printer does not contain the desired font.

9.6.1 Font selection by the creator

When a master is prepared, some knowledge of font names and properties is required by the creator. As we saw in Section 9.5, the creator can obtain in a metric master a list of all font names available on a printer or set of printers. While this list provides the names, it provides no additional information about a font that will help select among several available.

If a creator is producing a document entirely in one font, as for example a line printer listing of a program, then it can have the name of a font built in to it or can accept a font name as part of the command from the user that invokes the program. Since no font switching will be done during the course of the document, there is no need for more complex font selection

information. Usually a document of this kind is set in a fixed-pitch font, and there tend not to be very many such fonts available on a given printer.

In general a creator will need auxiliary information to help it select a font. Although there are many different kinds of programs that create masters in multiple fonts, a document preparation system is a nice example of a program that does so. Document preparation systems must be able to use a mix of body fonts, heading fonts, footnotes, italics, boldface, special-symbol fonts, and more. In each case, the user of the document preparation system somehow tells the system that he would like a font change and the document preparation system duly inserts appropriate font selection and font change code into the master being created. The creator must have some way of mapping the user's font-change requests into actual fonts that are known to be available on a printer.

In general, a creator will need to have access to information outside the scope of Interpress to help it select a font. That information can take the form of font profiles, font directories, document design specifications, or tables of various sorts. These things are part of the creator software and not part of Interpress software; they constitute part of the programming work that is necessary to make a document preparation system create Interpress masters.

For example, the document editor used to prepare this report labels text with named *styles*, and the editor allows its user to identify the name of the font that is to be used to show text in each style. This paragraph is in a style called "normal paragraph" and the subsection heading that follows it is in a style called "subsection heading." The mapping between these style names (which we specified when we composed the document) and actual font names is contained in a "style profile" file. Other creators may obtain font names from *configuration files* or *property sheets* or *document specification files*, which are text files that specify values for a number of parameters governing document printing, including font names. The details of how to store, represent, and access this auxiliary font information are entirely up to the creator, but in every case the information must describe the fonts to be used in an Interpress master.

9.6.2 Font approximation

A master extracts fonts from a printer's font library using the FINDFONT operator. Normally, the font requested by the master is available in the printer's library. But what happens if the font named in the call to FINDFONT is not available in the library? In this case, a printer doesn't just give up or summarily reject the master, but instead searches its font library looking for an *approximate* font to use as a substitute for the one it does not have. It is then able to print the document represented by the master, although the appearance of the product will not match precisely the images specified by the master.

The way in which an approximate font is located is not defined by Interpress. The intention is that a printer will retain, as an adjunct to its font library, some information that helps it to approximate font requests. It might, for example, keep for each font in the library a list of the fonts it approximates. Or the printer may have algorithms that it uses to find the approximate font, such as rules for examining the identifiers in the name of the requested font and matching them to names available in the font library. For example, if a font name includes an identifier that indicates a font's character set (see Section 9.2.3), the printer will limit its search for an approximate font to those fonts with the same character-set identifier.

When a printer doesn't have a font and is forced to choose an approximate substitute, it issues an "appearance warning" to alert the user to the fact that the appearance of the printed document is not precisely as specified in the master. Interpress does not specify how this warning is communicated to the user: it might be printed on a *break page* or *cover sheet* that precedes each printing job, or it might be typed out on a terminal attached to the printer, or an electronic message might be sent to the user over a communications system.

In practice, font approximation will seldom occur. This is because most masters will be constructed by creators that have access to a list of fonts available on the printer that will be used to print the master. The need for approximation arises if a master is prepared for one printer and then transmitted to another printer, one that has a different font library. Similarly, if in the course of maintenance and upgrades an obsolete font is removed from a printer's library, there may still be masters created and stored some time ago that request the font. If one of these masters is sent to the printer, an approximate font will have to be found, though often the approximation is to use the newer version of the missing font.

9.6.3 Font tuning

A printer may use an approximate font in order to increase image quality. This seemingly paradoxical situation arises when the printer's font has been "tuned" for the best possible appearance or legibility on that particular device [16]. While Interpress allows a single character operator to be scaled to arbitrary sizes, converting a standard definition into an array of dots to expose on a raster printer may introduce jagged edges, stroke-weight irregularities, and other artifacts that reduce the legibility of the font. To avoid these effects, a printer may retain separate character definitions for each size, with each definition carefully tuned to cater to the printer's imaging properties.

Font tuning may also alter very slightly the widths of characters so that each width is an integral number of dots on the printing device. Again, this modification is sometimes necessary to guarantee pleasant and legible juxtapositions of letterforms. While it might seem that even small deviations from the character widths anticipated by the creator would spoil the appearance of text, we'll see that the CORRECT operator, described in Section 10.4, can be used to compensate for any ill effects of font tuning.

9.6.4 Standardizing font libraries

It is clear that Interpress will work most smoothly if all printers have identical font libraries. As the differences between font libraries increase, the device independence of Interpress masters will decrease. Unfortunately, it is impractical to require that all font libraries be identical so that maximum device independence is achieved.

A practical solution to this problem is to strive to make the font libraries of all printers in one area be identical. Most masters are created, printed immediately, and discarded; moreover, when the master is created, the properties of the printer that will be used are known. A few masters are stored for printing later, but this later printing is usually done by one of a small number of printers: we can arrange that these printers all have identical font libraries. So if a particular organization, such as a hospital or a library, has a number of printers with identical font libraries, masters can be created and printed within the organization without difficulty.

Problems will increase when a document is transmitted "far" from its creation site in order to be printed. A printer at the Bank of America might have a very different set of fonts than one at the New York Times. The geographical distance between these two printers is unimportant, it is the fact that they are operated by two organizations with very different requirements that leads to the large "distance" between their font libraries. While the font approximation mechanism will permit any master to be printed, the approximation may yield a barely usable result.

An ideal solution to this problem would be to have a small number of truly standard fonts that some standards organization would define. These standard fonts would be available under standard names in Interpress, e.g., [*ansi, times*] and would adhere to standard character code sets and metrics. Then if every Interpress printer were to provide these standard fonts, a master could be guaranteed to print properly anywhere as long as it uses only the standard fonts. Even if only one or two fonts were standardized, the standard would be of enormous value. While such a standard does not now exist, Interpress' font mechanism will cheerfully accommodate any standard that is developed.

9.7 Summary

This section has collected a number of details about Interpress' font machinery. The main points are:

- Interpress allows fonts with arbitrary character sets to be used and intermixed freely in a master. *There is no such thing as a standard "Interpress character set."*
- Interpress allows virtually arbitrary font names by naming fonts with vectors of identifiers. Subject only to some restrictions that guarantee unique names (hierarchical names, explained in Section 11 and § 3.2.2), a font name may have an arbitrary number of identifiers that have arbitrary meanings.
- Characters are not a distinct type in Interpress. Rather, a character is defined by a sequence of standard Interpress operators. Not only does this make clear exactly what a character operator does, but it allows a master to define its own character operators to supplement or supplant those extracted from the printer's library.
- A character operator performs three functions: it makes a graphic image of the character, it alters the current position so as to prepare for the next character in sequence, and it invokes spacing corrections if necessary. The movement of the character position is restricted to one of two forms: movement by a fixed width or by a scaled width (*amplified characters*).
- The size, rotation, and position of character images are determined by the geometric transformations in force when the character operator is invoked. The size and rotation are usually controlled by both the transformation passed to MODIFYFONT *and* the current transformation at the time SHOW is called. Position is controlled by the current position when SHOW is called.
- Metric information, such as character widths, is provided to the creator by a *metric master.* The creator will usually reformat this information into a form more convenient for its use.

A good understanding of Interpress character operators and fonts is required before any complex typography is attempted. In the next section, we describe how the most common typographical effects may be achieved with Interpress.

10

Typography

Typography is the fine art of designing and placing letterforms to create a legible and pleasing effect. While many forms of computer output are so crude that worrying about typographic quality is futile, Interpress is able to describe documents of extremely high resolution and quality, and thus is able to represent documents with very fine typography.

This section is a catalog of "how to do it" recipes, that is, how various typographical effects can best be achieved in Interpress. It does not attempt to instruct the reader in the principles of good typography; such a task is far too ambitious for this report, and is moreover done better by others. The beginner is urged to consult *Designing with Type* [4] for a collection of practical hints for simple typography. For a thoughtful explanation of the visual and psychological principles behind typographic rules, see *First Principles of Typography* [13].

Interpress has been designed specifically to accommodate high-quality typography. Probably the single most important principle behind the design of Interpress is that *the printer makes no typographical decisions; all decisions are made by the creator.* Thus, for example, Interpress contains no notion of a "subscript," because if a master were to specify that a text string is a subscript, the printer would have to decide what type size to use and exactly where the subscript should appear, i.e., how far below the normal baseline the subscript's baseline should be. Instead, the creator places in the master instructions that control the precise size and placement of all text, whether sub/superscripts or normal text.

Interpress contains no automatic features or typesetting mechanisms and embodies no particular principles of typography within itself. It is instead a vehicle by which creators can represent the typographic decisions that they have made. Interpress can therefore represent documents that have been formatted according to many different sets of typographical rules or styles. As a corollary to this principle, there is nothing in Interpress that guarantees good typography; rather, there are mechanisms that permit and encourage it.

10.1 Typographic facilities in Interpress

Before exploring the various ways to achieve high-quality typography in Interpress, let us review the relevant facilities of Interpress. These are the facilities that are normally used to produce typeset text:

- *Letterform definitions*, expressed as character operators. These definitions might be extracted from a printer's font library or might be defined in the master itself. There are no restrictions imposed by Interpress on the shape of a letterform.
- *Positioning operators*, which are used to control the position of letters with respect to some coordinate system, and thus ultimately with respect to the page.
- *Geometric transformations* that can scale, rotate, and translate a letterform so that it can appear in an arbitrary size, rotation, and position on the page. Scaling and rotation are handled by both MODIFYFONT and the page coordinate system, while translation is handled by SETXY and SHOW.
- Additional *graphical operators* to define rules, underlines, strikethroughs, and the like. MASKVECTOR is an example of such an operator.

While this set of facilities is small, it is complete. This small set of primitives can be complete because Interpress places so few restrictions on how they can be used, either individually or in combination. There are no limits, for example, on the number of characters in a "line" of text; characters can appear at arbitrary positions; characters can be spaced closely or far apart, or may overlap; if necessary, a small letter can be placed inside a large one. It should be evident, in fact, that these primitives can specify a master in which any letterform is placed anywhere on the page, in any size, and rotated at any angle.

While a master can represent a document with high typographic quality, not all printers will be able to achieve such top-quality results. Some printers do not have sufficient resolution; some might use a coarse printing method that has specks or other forms of noise in the image; some printers place limits on the maximum complexity of a page or on the sizes of characters that can be printed. It is important to understand that these are limitations of a *printer*, not of *Interpress*—it is possible to build a printer that will print an arbitrary Interpress master perfectly. This section explains how to build Interpress masters that will print with a satisfactory appearance on less-than-perfect printers, as well as how to build Interpress masters that take advantage of the subtlety available only on the highest-quality printers.

10.2 Absolute and relative positioning

To set type, we must position characters on the page. A creator in possession of full knowledge of the font metrics of a printer could set type in Interpress by specifying exactly the x and y position of each character on the page, specifying character positions with the SETXY operator and printing characters with the SHOW operator.

All positioning is relative to some origin, as shown in Figure 10.1. If a character is positioned with respect to the origin of the page coordinate system, then we usually call that *absolute positioning*. If a character is positioned with respect to the origin of the previous character, then we usually call that *relative positioning*.

As an example of absolute positioning, let's repeat the substance of Example 3.2, but use a page coordinate system with 1/10 point units. In this example we will define a number of fonts in the preamble, which we will use in subsequent examples in this chapter. Throughout this chapter, we will assume the use of a page coordinate system in which the x axis direction points along the text baseline, in the direction in which characters are to be typeset. We will also assume that all characters run left-to-right, (i.e., have $widthX \geq 0$ and $widthY = 0$). In this first example, each character in the string "Interpress" is individually positioned.

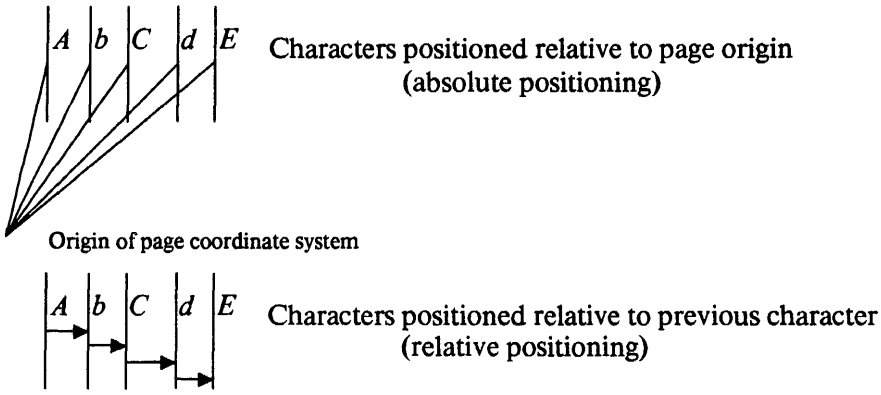


Figure 10.1. Absolute and relative character positioning.

The method of setting type shown below in Example 10.1 clearly shows Interpress' flexibility: each character can be positioned individually by the creator, without regard to any character width information used by the printer. A creator with full knowledge of the font metrics can position each character carefully and precisely to obtain the most pleasing effect. There are two drawbacks to this technique. First, it's not particularly device-independent, that is, the text will not look good if even a slight font approximation is made. The reason is that the spacing between characters is controlled by the master rather than by the character operators themselves (this is illustrated in Figure 10.2, line 4). Second, it leads to quite lengthy masters, which will consume too much storage space and slow down the computations performed by a printer printing the master.

```
--Example 10.1: Absolute character positioning (same image as Example 3.2)--
-- 0--    BEGIN {                          --begin preamble--
-- --                                      --define font 0 to be 10-point times--
-- 1--    [ xerox, xc82-0-0, times ] FINDFONT 100 SCALE MODIFYFONT 0 FSET
-- --                                      --define font 1 to be 10-point times italic--
-- 2--    [ xerox, xc82-0-0, timesitalic ] FINDFONT 100 SCALE MODIFYFONT 1 FSET
-- --                                      --define font 2 to be 12-point times--
-- 3--    [ xerox, xc82-0-0, times ] FINDFONT 120 SCALE MODIFYFONT 2 FSET
-- --                                      --define font 3 to be 8-point times--
-- 4--    [ xerox, xc82-0-0, times ] FINDFONT 80 SCALE MODIFYFONT 3 FSET
-- --                                      --define font 4 to be 18-point times--
-- 5--    [ xerox, xc82-0-0, times ] FINDFONT 180 SCALE MODIFYFONT 4 FSET
-- 6--    }                                --end preamble--
-- 7--    {                                --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT        --set page coordinate system to 1/10 point units--
-- 9--    4 SETFONT                        --sets the "current font" to 18-point times--
--10--    2088 6768 SETXY <I> SHOW         --prints 'I' at x=2.900 inches, y=9.4 inches--
--11--    2155 6768 SETXY <n> SHOW         --prints 'n' at x=2.993 inches, y=9.4 inches--
--12--    2257 6768 SETXY <t> SHOW         --prints 't' at x=3.135 inches, y=9.4 inches--
--13--    2313 6768 SETXY <e> SHOW         --prints 'e' at x=3.212 inches, y=9.4 inches--
--14--    2398 6768 SETXY <r> SHOW         --prints 'r' at x=3.331 inches, y=9.4 inches--
--15--    2465 6768 SETXY <p> SHOW         --prints 'p' at x=3.424 inches, y=9.4 inches--
--16--    2567 6768 SETXY <r> SHOW         --prints 'r' at x=3.565 inches, y=9.4 inches--
--17--    2634 6768 SETXY <e> SHOW         --prints 'e' at x=3.668 inches, y=9.4 inches--
--18--    2719 6768 SETXY <s> SHOW         --prints 's' at x=3.776 inches, y=9.4 inches--
--19--    2786 6768 SETXY <s> SHOW         --prints 's' at x=3.869 inches, y=9.4 inches--
--20--    }                                --end of the page body--
--21--    END                              --end of the master--
```

Interpress Electronic Printing Standard
Interpress Electronic Printing Standard
Interpress Electronic Printing Standard
Interpress Electronic Printing Standard

Figure 10.2. Font approximation

All four lines are meant to use 18 point Times Roman. The top two lines use relative positioning; the bottom two use absolute positioning. The first and third lines are printed using the exact font. The second and fourth lines show what happens if Times Roman is not available and Helvetica must be used instead. Note that the second line is too long and that the fourth line contains erratic spacing and character collisions.

Let us turn now to relative character positioning. The examples of typesetting in earlier sections have all relied on the character operators to achieve proper character-to-character spacing; SETXY is used only to position the first character in a line of text. In the example below, the entire string "Interpress" is printed with one call to SHOW. The position of the origin of the first character, "I", will be at $x=2.9$ inches, $y=9.4$ inches, the position set by the call to SETXY. The character operator for "I" will modify the current position to account for the width of the "I". The next character, "n", will be positioned so that its origin is at the modified current position. The character operator for "n" moves the current position still further, and so on. The effect is the same as that of Example 10.1, but the spacing is performed by the character operators rather than by explicit calls to SETXY. Besides yielding a more compact master, this technique does better in the presence of font approximation (see Figure 10.2, line 2): the intercharacter spacing will be correct for the font that is actually chosen, rather than for the font that the master requested.

```
--Example 10.2: implicit relative character spacing (same image as Example 3.2)--
--Lines 0 to 6 same as Example 10.1--
-- 7--    {                              --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT      --set page coordinate system to 1/10 point units--
-- 9--    4 SETFONT                      --sets current font to 18-point times--
--10--    2088 6768 SETXY                --sets current position to x=2.9 inches, y=9.4 inches--
--11--    <Interpress> SHOW              --prints 'Interpress'--
--12--    }                              --end of the page body--
--13--    END                            --end of the master--
```

This example still has a minor deficiency with respect to device independence. While the placement of the beginning of the "Interpress" string will always be at $x=2.9$ inches, $y=9.4$ inches, the intercharacter spacings will be those of the font actually used, so the end of the string might not appear in the right place, and the word will not be centered. We'll see in Section 10.6.9 how to fix this problem so that strings of text can be centered even in the presence of font approximation.

10.3 Measuring text

One of the most common calculations that a creator must make is to determine the length of a text string, i.e., the space that will be required to print it. Typographers refer to this distance as the *measure* of the text. The measure of a string is usually just the sum of the widths of the characters in the string, including the spaces. (Section 10.6.3, "Kerning," describes a situation in which the measure of a string is not the same as the sum of the character widths.) This measure calculation is used to decide where to break up a sequence of words into lines of a certain length, to decide where to position a string so that it will be centered, and so on.

The measure of a string can be computed in many different units. It is usually most convenient for the creator to compute the measure in the page coordinate system, which is the same system that will be used to position the string once its measure is known. The measure is thus the sum of the character widths as measured in the page coordinate system.

Since font metric information provides widths in the character coordinate system, we must convert from the character coordinate system to the page coordinate system when computing widths. This transformation is precisely the transformation that is provided to MODIFYFONT when a font is prepared, but we're interested only in the scaling component of the transformation. Let's define the *scale* of a font to be the SCALE portion of the transformation (or the product of the SCALE arguments if there is more than one). Thus the *scale* of font 0 in Example 10.1 is 100, the scale of font 2 is 120, and so on.

We can use the following *Measure* algorithm to compute the measure *m* in the page coordinate system of a string *s*. This algorithm assumes that all of the characters in *s* are horizontal, i.e., their *widthY* components are zero. The function *length*(*s*) is the number of characters in the string *s*, and *s*[*i*] denotes the *i*th character of the string *s*:

```
procedure Measure(s: string): real;
    var m: real; i: integer;
    begin
    m := 0;
    for i := 1 to length(s) do
        m := m + (widthX of s[i] from metric master)*(scale of s[i]);
    Measure := m
    end
```

This algorithm properly computes a string's measure even when characters from different fonts in different sizes are mixed in a string, as long as all of the characters are left-to-right (i.e., they have a *widthY* value of 0). Although only implied in the algorithm above, procedure *Measure* uses the *widthX* and *scale* information from the appropriate font for each character.

If you are familiar with the basic principles of floating-point arithmetic, you will realize that the computation of *m* in this algorithm will be subject to truncation error because relatively small numbers (character widths) are being added to relatively large numbers (the cumulative line measure). We cannot avoid this computational error, but in the next section we show how to take steps to ensure that it does not interfere with the appearance of the printed image.

In following sections, some of the examples use measure computations. To help you work through the examples, we have included some metric information from two real fonts. Table 10.1 shows metric information for some characters from hypothetical *xerox xc82-0-0 times* and *xerox xc82-0-0 timesitalic* fonts. As an example of the use of this table, consider computing the measure of the string "Interpress" as printed in Example 10.2. Since the scale of all the characters is 180, in this case we can sum the *widthX* values for the 10 characters in "Interpress" from Table 10.1, obtaining 4.254, and then multiply that sum by 180 to obtain 765.72, the measure of the string.

10.4 Positioning characters

The SHOW operator places character images into the page image at the *current position*, which is part of the imager state. The current position can be changed by executing an operator that provides new coordinates, which can be either *absolute* or *relative*. An absolute coordinate location is one that is specified with respect to a fixed origin; a relative coordinate location is one that is specified with respect to the current position. Good Interpress style requires that character positions be specified using a mixture of these two methods. Before explaining when to use which method, let's review the two methods.

The current position is set with absolute coordinates by the operator SETXY (§ 4.5):

⟨*x:* Number⟩ ⟨*y:* Number⟩ SETXY → ⟨⟩

where the current position is set to (*x*, *y*) after the coordinates have been transformed by the current transformation. Usually the coordinates passed to SETXY are in the page coordinate system, as illustrated in Example 10.2. Using SETXY, the current position can be set to an arbitrary location on the page.

The current position can also be set by giving relative coordinates with SETXYREL:

⟨*x:* Number⟩ ⟨*y:* Number⟩ SETXYREL → ⟨⟩

where the distance vector (*x*, *y*) is transformed by the current transformation and then *added* to the current position. Thus, the coordinates represent a *relative displacement* to the current position. Character operators use SETXYREL to move the current position to account for the character's width (see Examples 9.1 through 9.3).

It is important to understand that the *x* and *y* arguments to SETXY and SETXYREL will be transformed by the current transformation. Thus the coordinate system of the arguments to these operators is determined by the current transformation. The arguments to SETXY used in line 10 of Example 10.2 are evaluated in the page coordinate system. However, the arguments to SETXYREL in Examples 9.1 through 9.3 are evaluated in the character coordinate system. In both cases, these coordinate systems are established by the current transformation in effect when the SETXY and SETXYREL operators are called. If the current transformation involves only concatenations of SCALE and TRANSLATE transformations, the *x* coordinate passed to SETXY and SETXYREL will correspond to horizontal position or motion on the page (when held in the normal viewing orientation). However, if rotations are involved in the current transformation, the axis directions of arguments to the current-position operators will generally not correspond to the axis directions of the Interpress coordinate system.

Table 10.1 Width table for *times* and *times italic*

Character	*amplified*	*times widthX*	*times italic widthX*	Character	*amplified*	*times widthX*	*times italic widthX*
a	0	0.469	0.500	A	0	0.749	0.719
b	0	0.564	0.500	B	0	0.656	0.689
c	0	0.439	0.439	C	0	0.719	0.719
d	0	0.564	0.500	D	0	0.814	0.781
e	0	0.469	0.439	E	0	0.656	0.689
f	0	0.344	0.314	F	0	0.656	0.625
g	0	0.500	0.500	G	0	0.844	0.719
h	0	0.564	0.535	H	0	0.816	0.844
i	0	0.281	0.281	I	0	0.375	0.439
j	0	0.281	0.281	J	0	0.439	0.439
k	0	0.531	0.564	K	0	0.818	0.750
l	0	0.281	0.281	L	0	0.656	0.689
m	0	0.844	0.788	M	0	0.969	0.965
n	0	0.563	0.540	N	0	0.824	0.781
o	0	0.531	0.500	O	0	0.781	0.781
p	0	0.564	0.500	P	0	0.594	0.656
q	0	0.564	0.500	Q	0	0.781	0.781
r	0	0.375	0.375	R	0	0.751	0.719
s	0	0.375	0.375	S	0	0.564	0.625
t	0	0.314	0.318	T	0	0.656	0.656
u	0	0.564	0.535	U	0	0.813	0.781
v	0	0.500	0.439	V	0	0.719	0.719
w	0	0.719	0.656	W	0	1.000	1.000
x	0	0.500	0.564	X	0	0.751	0.719
y	0	0.500	0.469	Y	0	0.749	0.719
z	0	0.437	0.439	Z	0	0.656	0.719
0	0	0.500	0.500				
1	0	0.500	0.500				
2	0	0.500	0.500				
3	0	0.500	0.500				
4	0	0.500	0.500				
5	0	0.500	0.500				
6	0	0.500	0.500				
7	0	0.500	0.500	All characters have *widthY*=0.			
8	0	0.500	0.500				
9	0	0.500	0.500				
. (*period*)	0	0.248	0.250				
, (*comma*)	0	0.250	0.250				
(*space*)	1	0.250	0.250				

Warning: This table is provided only for working examples in this Introduction. Do not assume that a Xerox Interpress printer will have these fonts or these widths.

2. If the master has used relative positioning within the line (provided automatically by character operators), then intercharacter spacings will be correct but the total length of the line will not be right. This is illustrated in Figure 10.2, line 2.

If the creator seldom cared about the total length of the line, we could advise the creator to use relative positioning for maximum device independence. Unfortunately, when a creator justifies text lines, it cares *both* about intercharacter spacing, for maximum legibility, and about total line length, so that left and right edges of lines align. So the problem occurs sufficiently often to cause concern.

A hybrid technique that will preserve both character spacing and absolute positions is to use relative positioning to position characters with respect to one another, then periodically to resynchronize them with some absolute guideline. This technique is used by many high-quality electric clocks in large institutions, for example: the clock motor provides a reasonably accurate relative time, but once an hour the clock is resynchronized to a global time base that is broadcast from an expensive central source.

Interpress provides this kind of absolute synchronization of relative distances by means of the CORRECT operator. The CORRECT operator takes as its argument an Interpress body that prints a line of text, and takes from an imager variable the proper measure for that line. It will then adjust the positions of characters within the line (if necessary) to insure that the line fits exactly in the specified measure. Thus, when a creator cares where lines end as well as where they begin, CORRECT should be used so that the master will print reasonably even in the presence of font approximations. As you might expect, the inner workings of the CORRECT operator are more complex than those of SHOW; you should avoid using it in cases where it is not necessary, such as for making listings of computer files or for right-ragged text.

When CORRECT prints a line, it may need to contract or expand the spacing between characters within the line. In order to retain the text's legibility even in the presence of substantial mismatches between the font assumed by the creator and the one actually chosen by the printer, CORRECT distinguishes between two kinds of spacing:

1. CORRECTSPACE. The white space between words is called "CORRECTSPACE space." This space can safely be expanded a good deal without making the line too hard to read. However, it cannot be shrunk too much, or word spacings will not be evident. CORRECT will shrink this space only by the fraction *correctShrink* (an imager variable) before it takes other measures. That is, a distance *s* will never be shrunk to less than *s*correctShrink*.

2. CORRECTMASK. The spacing between two graphic shapes that represent characters is called "CORRECTMASK space." If a line must be shortened and the maximum shrinkage of the CORRECTSPACE space is insufficient to achieve the desired measure, characters will be moved closer together. This is clearly not good, but there is no other choice when a poor font approximation has been made.

The CORRECT operator can move characters around a little to correct the measure. This is one of the reasons why it's a good idea to use only relative positioning within a text line: if a character moves due to correction, you want the other things related to it (e.g., underlines, subscripts, superscripts, and accents) to move with it. You even want the word to its right to move with it.

While CORRECT can be used in many ways, here is a template that will work in almost all cases for printing a line of text. The CORRECT operator takes a single body as its argument.

```
--Example 10.3. CORRECT template. Assume inside a page body.--
--0--     tolerance 0 SETCORRECTTOLERANCE      --unless already done on this page--
--1--     correctShrinkValue 20 ISET           --unless already done on this page or--
-- --                                          --you like the default--
--2--     measure 0 SETCORRECTMEASURE          --unless it's already set to the right value--
--3--     x y SETXY                            --set starting position for text line--
--4--     {                                    --start CORRECT body--
--5--     --insert here calls to SHOW, SETFONT, SPACE, SETXYREL, SETXREL, SETYREL, etc.--
--6--     } CORRECT                            --end CORRECT body and call CORRECT--
```

CORRECT is the first Interpress operator we have used in this document that takes a body as its argument. "Body operators" of this sort are encoded somewhat differently than other operators—see § 2.5.2.

The first three lines (0–2) of the example above set various parameters. These lines can be omitted if the parameter has already been set to an appropriate value within the page body. The *tolerance* parameter instructs CORRECT how accurate the line measures must be. If the tolerance is large, CORRECT will not do as much work. For example, if the printer makes no font approximations, the text line should have almost exactly the right measure, but small roundoff errors can prevent it being exact. But if the line's measure is within the specified tolerance, CORRECT will not need to take any corrective action and the interpretation of the master will be correspondingly faster. If, for example, a measure error of 0.5 points can be tolerated, the master should set the tolerance accordingly (<5 0 SETCORRECTTOLERANCE> in the page coordinate system of Example 10.1 in which units are 1/10 point). The default value of *tolerance*, established at the beginning of each page body, is 0.

The *correctShrink* parameter limits the amount by which CORRECTSPACE space will be shrunk in order to shrink an entire line. Its default value, established at the beginning of each page body, is ½ (§ 4.2), so that a space will never become smaller than half its original size. There is no need to change this default if this value is acceptable.

The third parameter, the line's measure, might need to be changed more often, since different text lines on the page may have different measures. But sometimes the previous measure value can be used; for example, all the lines in this paragraph except the last one have the same measure.

In some cases, it is more convenient for the creator to set the measure after generating the Interpress commands to show the line of text. As a consequence, CORRECT allows the measure to be set within the CORRECT body. This corresponds to moving line 2 in Example 10.3 to between lines 5 and 6.

The *body* argument to CORRECT actually invokes Interpress operators that will print the text line. While there are no restrictions on the operators contained in this body, some guidelines should be observed to maximize the usefulness of CORRECT:

- Use only relative positioning (SETXYREL, SETXREL, SETYREL, SPACE). Use SPACE if the relative motion is generating "white space" that is to be treated as CORRECTSPACE space, such as space between words on a line. Use the other relative positioning operators if the spacing is not to be altered by CORRECT.
- Don't use simple relative motions to move backwards over characters already typeset within the line. If you need to use overstriking, or both super and subscripts on the same symbol, then somewhat more complex mechanisms are required (they are described later in this section).

- Any imager variables and frame elements changed within the CORRECT body (except the current position and the CORRECT line measure) will revert to their old values after CORRECT finishes. This behavior is characteristic of invoking bodies with DOSAVE, which is used by CORRECT, and is explained in Section 12.
- The body passed to CORRECT will be interpreted either once or twice, depending on whether or not the measure comes out right the first time. Don't try to put things in the body that depend on the number of times it will be executed.

It's now time to give a concrete example using the CORRECT template:

```
--Example 10.4: an example of the use of CORRECT--
--Lines 0 to 6 same as Example 10.1--
-- 7--    {                              --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT      --set page coordinate system to 1/10 point units--
-- 9--    5 0 SETCORRECTTOLERANCE        --set tolerance to 0.5 points--
--  --                                   --leave correctShrink set to its default value of 1/2--
--10--    2283 0 SETCORRECTMEASURE       --set measure to 3.17 inches--
--11--    720 6480 SETXY                 --set starting position for line, x=1 inch, y=9 inches--
--12--    {                              --start CORRECT body--
--13--    0 SETFONT                      --sets current font to 10-point times--
--14--    <An example of the CORRECT operator in > SHOW    --print text--
--15--    1 SETFONT                      --change font to 10-point times italic--
--16--    <Interpress.> SHOW
--17--    } CORRECT                      --invoke CORRECT--
--18--    }                              --end of the page body--
--19--    END                            --end of the master--
```

In this example, the measure is computed to be the width of the line:

An example of the CORRECT operator in *Interpress.*

using the widths obtained from Table 10.1 and the *Measure* algorithm. If you want to check your work, here's a breakdown of the width calculation:

Measure('An')	131.2
Measure('example')	359.6
Measure('of')	87.5
Measure('the')	134.7
Measure('CORRECT')	503.3
Measure('operator')	362.8
Measure('in')	84.4
Measure('Interpress.')	444.5
7 Spaces @ 25 each	175.0
Total measure:	**2283.0**

CORRECT is not intended to be used to achieve proper justification, but rather to force a line of text to be its intended length even in the presence of font approximation. The key here is that "intended length" is the length of the line computed assuming perfect font matches. If you want to justify a line of text, use the techniques explained in the next section.

This section and the examples present only simple uses of CORRECT. A detailed description of the operation of CORRECT is presented in § 4.10.

Table 10.1 Width table for *times* and *times italic*

Character	*amplified*	*times widthX*	*times italic widthX*	Character	*amplified*	*times widthX*	*times italic widthX*
a	0	0.469	0.500	A	0	0.749	0.719
b	0	0.564	0.500	B	0	0.656	0.689
c	0	0.439	0.439	C	0	0.719	0.719
d	0	0.564	0.500	D	0	0.814	0.781
e	0	0.469	0.439	E	0	0.656	0.689
f	0	0.344	0.314	F	0	0.656	0.625
g	0	0.500	0.500	G	0	0.844	0.719
h	0	0.564	0.535	H	0	0.816	0.844
i	0	0.281	0.281	I	0	0.375	0.439
j	0	0.281	0.281	J	0	0.439	0.439
k	0	0.531	0.564	K	0	0.818	0.750
l	0	0.281	0.281	L	0	0.656	0.689
m	0	0.844	0.788	M	0	0.969	0.965
n	0	0.563	0.540	N	0	0.824	0.781
o	0	0.531	0.500	O	0	0.781	0.781
p	0	0.564	0.500	P	0	0.594	0.656
q	0	0.564	0.500	Q	0	0.781	0.781
r	0	0.375	0.375	R	0	0.751	0.719
s	0	0.375	0.375	S	0	0.564	0.625
t	0	0.314	0.318	T	0	0.656	0.656
u	0	0.564	0.535	U	0	0.813	0.781
v	0	0.500	0.439	V	0	0.719	0.719
w	0	0.719	0.656	W	0	1.000	1.000
x	0	0.500	0.564	X	0	0.751	0.719
y	0	0.500	0.469	Y	0	0.749	0.719
z	0	0.437	0.439	Z	0	0.656	0.719
0	0	0.500	0.500				
1	0	0.500	0.500				
2	0	0.500	0.500				
3	0	0.500	0.500				
4	0	0.500	0.500				
5	0	0.500	0.500				
6	0	0.500	0.500				
7	0	0.500	0.500	All characters have *widthY*=0.			
8	0	0.500	0.500				
9	0	0.500	0.500				
. (*period*)	0	0.248	0.250				
, (*comma*)	0	0.250	0.250				
(*space*)	1	0.250	0.250				

Warning: This table is provided only for working examples in this Introduction. Do not assume that a Xerox Interpress printer will have these fonts or these widths.

There are some variants of SETXYREL that are commonly used for typographical purposes:

⟨*x:* Number⟩ SETXREL → ⟨⟩
where the effect is equivalent to ⟨*x* 0 SETXYREL⟩, i.e., the current position is displaced in the *x* direction but not in the *y* direction. The calls to SETXYREL in Examples 9.1 through 9.3 could be replaced with corresponding calls to SETXREL.

⟨*y:* Number⟩ SETYREL → ⟨⟩
where the effect is equivalent to ⟨0 *y* SETXYREL⟩, i.e., the current position is displaced in the *y* direction but not in the *x* direction.

⟨*x:* Number⟩ SPACE → ⟨⟩
where the effect on the current position is the same as ⟨*x* 0 SETXYREL⟩, but the current position might be corrected slightly if necessary to account for font approximation.

As we'll see below, SPACE is most often used for inter-word spaces in justified text.

10.4.1 When to use absolute and relative positioning

While there are no hard and fast rules about the use of positioning operators, good Interpress style leads to the following guidelines:

- Use relative positioning when the relationship between two adjacent objects is important. Characters within a word, or words within a line should be positioned relative to each other. This is why all character operators use relative positioning.
- Use absolute positioning to locate unrelated or loosely-related objects on the page. Thus the beginning of an entire line of text or an entry in a column of a table should be positioned with absolute positioning.
- Never allow more than about 250 relative positioning operators between absolute positioning operators. (Remember that each invocation of a character operator calls a relative positioning operator once.) The reason is that small numerical errors usually accompany each relative addition to the current position. If too many of these errors accumulate, the positioning error will be noticeable. Precision rules that printers must obey (§ 5.2) yield unnoticeable errors as long as fewer than 250 relative motions are used.

Relative positioning of adjacent objects makes an Interpress master more device-independent, that is, more immune to ill effects of font approximation. Thus Example 10.1, which uses absolute positioning of characters, will print badly if a poor font approximation is made, because characters will collide or overlap if the widths of the actual character operators used are greater than those assumed when the master was created. Example 10.2, which prints the same text, is more immune to font approximation.

By contrast, absolute positioning should be used to achieve "global" or long-distance positioning relationships. For example, if several lines of text are to align at the left (so-called *left flush*), it is best to set the starting position of each line with an absolute position, as in Example 3.7. While one could imagine replacing line 6 of that example with an appropriate call to SETXYREL, the resulting position might not align with the first text line if the character widths of the font actually used by SHOW in line 5 were not the same as the ones assumed when calculating the arguments to SETXYREL.

When you are deciding whether to use absolute or relative positioning, ask yourself the question "If the printer doesn't have the font requested and uses an approximate font in which characters have different widths than those of the requested font, what will happen to the printed result?" Usually you will be able to decide how you want the image to degrade when font approximations occur.

10.4.2 Precision in character widths

It is important to use sufficient precision in calculating the measure of text lines (c.f. the *Measure* routine in Section 10.3). At a viewing distance of about 12 inches, the eye can detect a 1/400 inch positioning difference between two objects sufficiently close together. For example, you should be able to see that one of the vertical bars in Figure 10.3 is positioned to the right of the others.

Figure 10.3. A slight positioning difference

The bar in the middle row is positioned about $1/350^{th}$ of an inch to the right of the other bars. While you may feel that this example is contrived to make a tiny positioning error noticeable, visible errors arise more frequently than you might expect. Although absolute positioning avoids such errors easily, avoiding positioning errors with relative positioning requires precise calculation.

When the measure of a line is computed, many character widths are summed, so that small errors will accumulate. If the cumulative error is more than 1/400 inch, it might produce a visible misalignment. The Interpress precision rules suggest that up to 250 character widths can be safely summed. To meet these criteria, you should be sure that the absolute error in each character width is less than $(1/2)\times(1/400)\times(1/250)=1/200{,}000$ inch. For 10-point characters, that means keeping character widths using a representation that is precise to about one part in 30,000 ($10/72*(1/30{,}000)\approx 1/200{,}000$). For larger characters, more precision would normally be required, but it is probably safe to argue that when larger characters are positioned, far fewer than 250 roundoff errors will accumulate before a line break or other absolute positioning occurs. These precision requirements can be met by most computer floating-point representations, including the IEEE standard [3, 30].

10.4.3 Spacing correction

The discussion above has slighted an important issue, namely that you might be concerned about proper intercharacter spacing *and also* about the total length of a line when font approximations are made. As always, if no font approximations occur, the creator knows exactly the intercharacter spacings that the printer will use and can therefore anticipate exactly where the line will end by summing the widths of all the characters. But if a font approximation occurs, one of two things happens:

1. If the master has used absolute positioning of characters, then the total length of the line will be right, but the intercharacter spacings will not be appropriate for the font actually used. This is illustrated in Figure 10.2, line 4.

2. If the master has used relative positioning within the line (provided automatically by character operators), then intercharacter spacings will be correct but the total length of the line will not be right. This is illustrated in Figure 10.2, line 2.

If the creator seldom cared about the total length of the line, we could advise the creator to use relative positioning for maximum device independence. Unfortunately, when a creator justifies text lines, it cares *both* about intercharacter spacing, for maximum legibility, and about total line length, so that left and right edges of lines align. So the problem occurs sufficiently often to cause concern.

A hybrid technique that will preserve both character spacing and absolute positions is to use relative positioning to position characters with respect to one another, then periodically to resynchronize them with some absolute guideline. This technique is used by many high-quality electric clocks in large institutions, for example: the clock motor provides a reasonably accurate relative time, but once an hour the clock is resynchronized to a global time base that is broadcast from an expensive central source.

Interpress provides this kind of absolute synchronization of relative distances by means of the CORRECT operator. The CORRECT operator takes as its argument an Interpress body that prints a line of text, and takes from an imager variable the proper measure for that line. It will then adjust the positions of characters within the line (if necessary) to insure that the line fits exactly in the specified measure. Thus, when a creator cares where lines end as well as where they begin, CORRECT should be used so that the master will print reasonably even in the presence of font approximations. As you might expect, the inner workings of the CORRECT operator are more complex than those of SHOW; you should avoid using it in cases where it is not necessary, such as for making listings of computer files or for right-ragged text.

When CORRECT prints a line, it may need to contract or expand the spacing between characters within the line. In order to retain the text's legibility even in the presence of substantial mismatches between the font assumed by the creator and the one actually chosen by the printer, CORRECT distinguishes between two kinds of spacing:

1. CORRECTSPACE. The white space between words is called "CORRECTSPACE space." This space can safely be expanded a good deal without making the line too hard to read. However, it cannot be shrunk too much, or word spacings will not be evident. CORRECT will shrink this space only by the fraction *correctShrink* (an imager variable) before it takes other measures. That is, a distance *s* will never be shrunk to less than *s*correctShrink*.

2. CORRECTMASK. The spacing between two graphic shapes that represent characters is called "CORRECTMASK space." If a line must be shortened and the maximum shrinkage of the CORRECTSPACE space is insufficient to achieve the desired measure, characters will be moved closer together. This is clearly not good, but there is no other choice when a poor font approximation has been made.

The CORRECT operator can move characters around a little to correct the measure. This is one of the reasons why it's a good idea to use only relative positioning within a text line: if a character moves due to correction, you want the other things related to it (e.g., underlines, subscripts, superscripts, and accents) to move with it. You even want the word to its right to move with it.

While CORRECT can be used in many ways, here is a template that will work in almost all cases for printing a line of text. The CORRECT operator takes a single body as its argument.

```
--Example 10.3. CORRECT template. Assume inside a page body.--
--0--    tolerance 0 SETCORRECTTOLERANCE      --unless already done on this page--
--1--    correctShrinkValue 20 ISET           --unless already done on this page or--
-- --                                         --you like the default--
--2--    measure 0 SETCORRECTMEASURE          --unless it's already set to the right value--
--3--    x y SETXY                            --set starting position for text line--
--4--    {                                    --start CORRECT body--
--5--    --insert here calls to SHOW, SETFONT, SPACE, SETXYREL, SETXREL, SETYREL, etc.--
--6--    } CORRECT                            --end CORRECT body and call CORRECT--
```

CORRECT is the first Interpress operator we have used in this document that takes a body as its argument. "Body operators" of this sort are encoded somewhat differently than other operators—see § 2.5.2.

The first three lines (0−2) of the example above set various parameters. These lines can be omitted if the parameter has already been set to an appropriate value within the page body. The *tolerance* parameter instructs CORRECT how accurate the line measures must be. If the tolerance is large, CORRECT will not do as much work. For example, if the printer makes no font approximations, the text line should have almost exactly the right measure, but small roundoff errors can prevent it being exact. But if the line's measure is within the specified tolerance, CORRECT will not need to take any corrective action and the interpretation of the master will be correspondingly faster. If, for example, a measure error of 0.5 points can be tolerated, the master should set the tolerance accordingly (<5 0 SETCORRECTTOLERANCE> in the page coordinate system of Example 10.1 in which units are 1/10 point). The default value of *tolerance*, established at the beginning of each page body, is 0.

The *correctShrink* parameter limits the amount by which CORRECTSPACE space will be shrunk in order to shrink an entire line. Its default value, established at the beginning of each page body, is ½ (§ 4.2), so that a space will never become smaller than half its original size. There is no need to change this default if this value is acceptable.

The third parameter, the line's measure, might need to be changed more often, since different text lines on the page may have different measures. But sometimes the previous measure value can be used; for example, all the lines in this paragraph except the last one have the same measure.

In some cases, it is more convenient for the creator to set the measure after generating the Interpress commands to show the line of text. As a consequence, CORRECT allows the measure to be set within the CORRECT body. This corresponds to moving line 2 in Example 10.3 to between lines 5 and 6.

The *body* argument to CORRECT actually invokes Interpress operators that will print the text line. While there are no restrictions on the operators contained in this body, some guidelines should be observed to maximize the usefulness of CORRECT:

- Use only relative positioning (SETXYREL, SETXREL, SETYREL, SPACE). Use SPACE if the relative motion is generating "white space" that is to be treated as CORRECTSPACE space, such as space between words on a line. Use the other relative positioning operators if the spacing is not to be altered by CORRECT.
- Don't use simple relative motions to move backwards over characters already typeset within the line. If you need to use overstriking, or both super and subscripts on the same symbol, then somewhat more complex mechanisms are required (they are described later in this section).

- Any imager variables and frame elements changed within the CORRECT body (except the current position and the CORRECT line measure) will revert to their old values after CORRECT finishes. This behavior is characteristic of invoking bodies with DOSAVE, which is used by CORRECT, and is explained in Section 12.
- The body passed to CORRECT will be interpreted either once or twice, depending on whether or not the measure comes out right the first time. Don't try to put things in the body that depend on the number of times it will be executed.

It's now time to give a concrete example using the CORRECT template:

```
--Example 10.4: an example of the use of CORRECT--
--Lines 0 to 6 same as Example 10.1--
-- 7--    {                              --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT      --set page coordinate system to 1/10 point units--
-- 9--    5 0 SETCORRECTTOLERANCE        --set tolerance to 0.5 points--
--  --                                   --leave correctShrink set to its default value of 1/2--
--10--    2283 0 SETCORRECTMEASURE       --set measure to 3.17 inches--
--11--    720 6480 SETXY                 --set starting position for line, x=1 inch, y=9 inches--
--12--    {                              --start CORRECT body--
--13--    0 SETFONT                      --sets current font to 10-point times--
--14--    <An example of the CORRECT operator in > SHOW    --print text--
--15--    1 SETFONT                      --change font to 10-point times italic--
--16--    <Interpress.> SHOW
--17--    } CORRECT                      --invoke CORRECT--
--18--    }                              --end of the page body--
--19--    END                            --end of the master--
```

In this example, the measure is computed to be the width of the line:

An example of the CORRECT operator in *Interpress.*

using the widths obtained from Table 10.1 and the *Measure* algorithm. If you want to check your work, here's a breakdown of the width calculation:

Measure('An')	131.2
Measure('example')	359.6
Measure('of')	87.5
Measure('the')	134.7
Measure('CORRECT')	503.3
Measure('operator')	362.8
Measure('in')	84.4
Measure('Interpress.')	444.5
7 Spaces @ 25 each	175.0
Total measure:	**2283.0**

CORRECT is not intended to be used to achieve proper justification, but rather to force a line of text to be its intended length even in the presence of font approximation. The key here is that "intended length" is the length of the line computed assuming perfect font matches. If you want to justify a line of text, use the techniques explained in the next section.

This section and the examples present only simple uses of CORRECT. A detailed description of the operation of CORRECT is presented in § 4.10.

10.5 Justifying text

There are a number of techniques that can be used to typeset justified lines of text in Interpress. In all cases, the creator is concerned both with intercharacter spacing and with total line measure, so the CORRECT operator should be used to force the justified appearance even when font approximations must be made.

The first part of any justification mechanism determines where to break a line, either at a space between words or at a hyphen inserted in a word. The details of this decision depend a great deal on the typographical rules and styles that the creator wants to uphold. A basic computation necessary to all rules is the determination of the measure of a word, i.e., the linear space that a word will occupy when printed in its prescribed font. This calculation is performed by the *Measure* algorithm given in Section 10.3.

The second part of a justification algorithm consists of computing the spacings between words so that the entire line will have the right measure. Again, typographical styles differ. Simple justification algorithms will make all inter-word spaces equal, while more complex algorithms might vary the widths slightly in order to avoid unpleasant visual effects like *rivers* in blocks of justified text.

10.5.1 Justifying with the SPACE operator

The most general way to typeset justified text in Interpress is to use relative positioning of the current position to specify the desired inter-word spacings. The SPACE operator is used for this purpose, since it both achieves relative motion and interacts properly with CORRECT (see Section 10.4.3).

```
--Example 10.5: use of the SPACE operator in justification--
--Lines 0 to 6 same as Example 10.1--
-- 7--    {                              --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT      --set page coordinate system to 1/10 point units--
-- 9--    5 0 SETCORRECTTOLERANCE        --set tolerance to 0.5 points--
--  --                                   --leave correctShrink set to its default value of 1/2--
--10--    2520 0 SETCORRECTMEASURE       --set measure to 3.5 inches--
--11--    720 6480 SETXY                 --set starting position for line, x=1 inch, y=9 inches--
--12--    {                              --start CORRECT body--
--13--    0 SETFONT                      --sets current font to 10-point times--
--14--    <An> SHOW 59 SPACE             --first word and trailing space--
--15--    <example> SHOW 59 SPACE
--16--    <of> SHOW 59 SPACE
--17--    <the> SHOW 59 SPACE
--18--    <CORRECT> SHOW 59 SPACE
--19--    <operator> SHOW 59 SPACE
--20--    <in> SHOW 58 SPACE
--21--    1 SETFONT                      --change font to 10-point times italic--
--22--    <Interpress.> SHOW
--23--    } CORRECT                      --invoke CORRECT--
--24--    }                              --end of the page body--
--25--    END                            --end of the master--
```

Example 10.5 shows a master that prints the same text as in Example 10.4. Suppose that the line of text is supposed to occupy 21 picas, which is 3.5 inches, or 2520 distance units in the page coordinate system that we are using (1/10 points). We use the *Measure* algorithm to determine that the total width of the words in the text, not counting spaces, is 2108 units in the page coordinate system. Thus 2520−2108, or 412 units must be spread among the 7 inter-word spaces. If each space is 58.86 units wide, the 7 spaces will total 412.02 units, which is so

close that the difference will not be apparent on the page. Alternatively, 6 of the 7 spaces can be made 59 units wide, and the remaining one 58 units wide, for a total of exactly 412 units. The second approach is used in the example, since it requires only integers to appear in the master.

Note, in this example, that the measure specified to CORRECT (line 10) has been changed to reflect the length of the properly justified line. Observe that two separate mechanisms are at work here. The SPACE operators achieve justification. The CORRECT operator insures that the line length will be correct even if a font approximation is made.

It is clear that this technique allows considerable flexibility in the allocation of space between words, e.g., interword spaces need not be equal. The only drawback of this scheme is that the master gets rather bulky because each word is passed to SHOW individually. A more compact but less flexible technique is described in the next section.

10.5.2 Justifying with amplified spaces

Example 10.5 could be shortened considerably if the master were able to instruct Interpress to alter the width of the "space" character while the line of text is printed. If the space character were altered to have a width of 58.86 units, just for this line, then words and spaces alike could be passed to SHOW and the line would come out justified.

Interpress achieves this effect by using *amplifying spaces*. An amplifying space is a character operator that makes no mark upon the page and that has a varying width, depending on the value of some variable. This is easy to do in Interpress because every character is an operator, and if we choose to make the space operator a bit more complex than the rest, that complexity is completely taken in stride by the SHOW operator. Recall from Section 9.3.3 that amplifying characters multiply their *widthX* and *widthY* values by the imager variable *amplifySpace* to determine the relative positioning to perform. So by using amplifying spaces for inter-word spaces and by setting *amplifySpace* correctly, the inter-word spaces can be set so that the line is justified.

In order to determine the value of *amplifySpace* that must be used to justify a line of text, it is necessary to measure the text in such a way that the widths of amplified and normal characters are kept separate. The computation might be expressed as follows ($s[i]$ is the ith character of the string s, and *length*(s) is the number of characters in the string s):

```
var m,ma: real; i: integer;
begin
m := 0; ma := 0;
for i := 1 to length(s) do
    if (s[i] is amplified, as determined from the metric master) then
        ma := ma + (widthX of s[i] from metric master)*(scale of s[i])
    else
        m := m + (widthX of s[i] from metric master)*(scale of s[i]);
```

This modification sums in m the width of the normal characters and in ma the width of the amplified characters. The total measure of the string, when printed by Interpress, will thus be $totalMeasure = m + ma * amplifySpace$.

If we know what the total width of the line, *totalMeasure*, is supposed to be, we can easily solve this equation for the setting of *amplifySpace* that should be put in the master. In our example above, *totalMeasure*=2520, and the values of *m* and *ma* for the entire line of text are *m*=2108, *ma*=175 (note from Table 10.1 that the normal "space" character is amplifying). Thus we compute that *amplifySpace*=2.3543, or, if you prefer, the rational fraction 412/175.

We can now rework Example 10.5 to use amplified spaces:

```
--Example 10.6: Justification using amplified spaces--
--Lines 0 to 6 same as Example 10.1--
-- 7--    {                              --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT      --set page coordinate system to 1/10 point units--
-- 9--    5 0 SETCORRECTTOLERANCE        --set tolerance to 0.5 points--
--  --                                   --leave correctShrink set to its default value of 1/2--
--10--    2520 0 SETCORRECTMEASURE       --set measure to 3.5 inches--
--11--    720 6480 SETXY                 --set starting position for line, x=1 inch, y=9 inches--
--12--    {                              --start CORRECT body--
--13--    2.3543 18 ISET                 --set amplifySpace--
--14--    0 SETFONT                      --sets current font to 10-point times--
--15--    <An example of the CORRECT operator in > SHOW    --print text--
--16--    1 SETFONT                      --change font to 10-point times italic--
--17--    <Interpress.> SHOW
--18--    } CORRECT                      --invoke CORRECT--
--19--    }                              --end of the page body--
--20--    END                            --end of the master--
```

Note that the setting of *amplifySpace* will not persist beyond the execution of the CORRECT operator on line 18, because as we remarked earlier, non-persistent imager variables changed inside the body argument of CORRECT revert to their former values when CORRECT terminates.

Justification with amplified spaces is recommended in all but the highest quality typographic applications. As demonstrated in Example 10.6, justification can be achieved without appreciably lengthening the master or slowing its interpretation. Amplified spaces can be used together with the SPACE operator, explained in Section 10.5.1, to achieve more complex effects. A particularly important combination is to use amplified spaces for inter-word space, but to use the SPACE operator for inter-sentence space.

10.6 Other typographical effects

This section provides some hints and examples for achieving various different typographical effects in Interpress. This list is not exhaustive, but should provide some insights into how to achieve additional effects.

10.6.1 Flush left, ragged right

Flush left, right-ragged text is handled very simply in Interpress. The starting position of each line is specified with SETXY, followed by commands to show the line. It is usually not necessary to use CORRECT for right-ragged text. However, if the right edge must not move too far in the presence of font approximations, CORRECT could be required. For example, if ragged text is set inside a box, with rules around it, CORRECT can insure that text will never cross the rules. Also, the last line of a justified paragraph may need to use CORRECT to be sure that, in the presence of font approximations, the right edge of the text does not extend beyond the right edge of the justified lines.

10.6.2 Flush right, ragged left

If lines of type are to align at their right sides, CORRECT must be used to be sure that font approximations do not cause the line length to vary. The starting position of the line is specified with SETXY and the line's total length (distance from starting position to right alignment point) is specified as the measure to CORRECT.

If a line of text has no spaces in it, a space will need to be inserted at the left of the line in order for this technique to work.

10.6.3 Kerning

Often the most pleasing spacing of a pair of characters is not the width associated with the first character. For example, if the two characters A and V are printed using normal spacing (i.e., A's width), they will appear to be too far apart because the right side of the A and the left side of the V are parallel. Kerning is a technique for altering intercharacter spacing of pairs of characters to achieve a better appearance, illustrated in Figure 10.4. The figure uses capital letters because kerning is much more visible in capital letters; lowercase letters may be kerned as well.

FAR AWAY Text set with standard letter spacing

FAR AWAY Text set with kerned letter spacing

Figure 10.4. Kerning

Kerning can be specified explicitly in the master using relative positioning. In this case, SETXREL is appropriate. To print the sequence "AV", we might use the sequence <<A> SHOW −2 SETXREL <V> SHOW>. The call to SETXREL will move the current position to the left slightly so that the V is closer to the A than it would be otherwise. Don't forget that kerning adjustments will alter the measure of a line; this must be taken into account by the *Measure* algorithm.

The size of the kerning adjustment must be determined by the creator. Suggested kerning data might be available from the printer as part of the font metrics (§ 4.9.3) or might be derived by the creator from other information about the fonts it uses.

It is important to use relative positioning to achieve kerning so that the master does not depend on highly accurate correspondence between character widths known to the creator and those that will be used by the printer. If CORRECT alters slightly the position of the "A", the position of the "V" will move as well. (Note that we use SETXREL for kerning, rather than SPACE, because we don't want CORRECT to alter this relative motion at all.)

If kerning is used extensively, the Interpress master can become quite bulky because of the numerous calls to SHOW and SETXREL. The bulk can be reduced by encoding character indices and kern offsets alternately in a single vector and using the SHOWANDXREL operator (see § 4.4.6). The example above could be encoded as <[65, 126, 86] SHOWANDXREL>.

The Interpress conventions for character operators do not do kerning automatically for several reasons: current typographic practices discourage kerning except for occasional display type and very high quality text; the amount of kerning adjustment is not always a unique property of a font, but will differ for different applications; and if character pairs of different sizes or from different fonts are to be kerned, the number of different kerning adjustments that must be stored is too large to require all Interpress imagers to save.

10.6.4 Letterspacing

In some cases, especially in headlines or "display type," spacing between characters must be increased beyond the normal width spacing. This is called *letterspacing*, illustrated in Figure 10.5.

letterspacing not applied
letterspacing applied

Figure 10.5: Letterspacing

Letterspacing can also be used to avoid "collisions" between characters: e.g., *f*) rather than *f*). Letterspacing is a form of kerning, except that the spacing corrections tend to be positive rather than negative and are uniformly applied to all spaces between letters in a word. All of the remarks devoted to kerning in the previous section apply to letterspacing.

10.6.5 Accents and diacritical marks

The handling of accents and diacritical marks is controlled by designers of character sets and fonts. Three methods are common:

- The character set contains a separate entry for each combination of character and mark. Thus é would be represented as a single character in the character set.
- The character set contains separate characters for each accent and diacritical mark, whose widths are zero. The accents are designed and positioned so as to have the correct relationship to a character placed next after the accent: to generate é the master would call <'e> SHOW. This technique is useful for modest-quality fonts, such as those available on automatic typewriters, in which positioning accuracy is sufficiently poor that the detailed location of the accent mark over the character does not matter much.
- The font has separate accent characters, as in the second case, but the master must specify the correct positioning of the accent.

The first two techniques can be used without any special treatment of the accents or characters. For the last technique, the accent or diacritical mark can be positioned with relative adjustments to the current position, but usually care must be taken that including the accent does not disrupt the spacing of characters in a word. That is, if a relative motion is used to position the accent properly, a compensating inverse relative motion is used to return the current position to its previous position. For example, to show the text "Bézier" and to position the acute accent carefully, we might use:

```
--Example 10.7: positioning an accent character with SETXYREL--
--0--     <B> SHOW
--1--     2 3 SETXYREL          --adjust position for accent--
--2--     <'> SHOW              --print accent, which has widthX=0--
--3--     -2 -3 SETXYREL        --return to position after line 0, assuming accent widths=0--
--4--     <ezier> SHOW
```

An alternative way to proceed is to save the current position on the Interpress operand stack and then restore it:

```
--Example 10.8: positioning an accent character with SETXYREL and IGET/ISET--
--0--     <B> SHOW
--1--     0 IGET 1 IGET         --save current position on the stack--
--2--     2 3 SETXYREL          --adjust position for accent--
--3--     <'> SHOW              --print accent--
--4--     1 ISET 0 ISET         --restore current position--
--5--     <ezier> SHOW
```

As is the case for kerning, the amount by which an accent is offset must be determined from exogenous information about the font.

There is actually a fourth method for dealing with accents, namely to use ligatures. For example, in ISO 6937, the single graphic character ä is specified by two character codes in sequence: first, one for ¨, and then one for a. The graphic symbol that is printed, ä, is not just an a overprinted with a ¨, but a graphic designed for just this situation. This approach allows other accented characters, such as ü or ö to be represented without requiring an explosion of character codes in the character set. This situation is analogous to conventional ligatures, e.g., f and i, in which the two characters can be printed sequentially for modest-quality applications, or a signle specially-designed ligature letterform can be used instead. Interpress will not substitute ligatures for two-character sequences automatically, but instead requires the creator to do this job (Section 9.3.4). For diacritical marks as well as for conventional ligatures, the metric master can provide the creator with the necessary ligature information (Section 9.4).

10.6.6 Underlines

While high-quality typography frowns on underlines in favor of *italics*, underlines are often used in word-processing applications and other low-quality documents. In Interpress, an underline is actually a stroke drawn under some text. However, two Interpress operators help to position the underline properly (§ 4.8.2):

‹› STARTUNDERLINE → ‹›

where the current position is remembered in the imager variable *underlineStart*. The STARTUNDERLINE operator is used to record where an underline is to start.

‹*dy:* Number› ‹*h:* Number› MASKUNDERLINE → ‹›

where an underline in the *x* direction is drawn from the position recorded by STARTUNDERLINE to the current position. The thickness of the underline is set by the parameter *h* and the top of the line will be displaced *dy* units below the current position.

The following master is like Example 10.7, but the word CORRECT will be underlined with a 1-point rule ($h=10$), displaced 2 points below the baseline ($dy=20$):

```
--Example 10.9: underlining--
--Lines 0 to 6 same as Example 10.1--
-- 7--    {                              --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT      --set page coordinate system to 1/10 point units--
-- 9--    5 0 SETCORRECTTOLERANCE        --set tolerance to 0.5 points--
--  --                                   --leave correctShrink set to its default value of 1/2--
--10--    2520 0 SETCORRECTMEASURE       --set measure to 3.5 inches--
--11--    720 6480 SETXY                 --set starting position for line, x=1 inch, y=9 inches--
--12--    {                              --start CORRECT body--
--13--    2.3543 18 ISET                 --set amplifySpace--
```

```
--14--    0 SETFONT                        --sets current font to 10-point times--
--15--    <An example of the > SHOW
--16--    STARTUNDERLINE                   --record starting position for underline--
--17--    <CORRECT> SHOW                   --print text to be underlined--
--18--    20 10 MASKUNDERLINE              --underline--
--19--    < operator in > SHOW             --print text--
--20--    1 SETFONT                        --change font to 10-point times italic--
--21--    <Interpress.> SHOW
--22--    } CORRECT                        --invoke CORRECT--
--23--    }                                --end of the page body--
--24--    END                              --end of the master--
```

Note that MASKUNDERLINE generates only a single stroke. If several lines of text are to be underlined, each one will require a STARTUNDERLINE, MASKUNDERLINE pair.

If instead of underlining, we wanted to place a rule through the middle of the word (a so-called "strikethrough," which looks like this: ~~CORRECT~~), we could use a negative value for *dy* so that the rule would be placed above the baseline.

The underline operators are designed to place the underline relative to the position of the text when it is actually printed: if CORRECT makes small changes to the position of text characters, the underline will move as well, so as to retain its position relative to the characters.

10.6.7 Disabling spacing correction

In some cases, it is necessary to disable the effects of spacing correction in part, but not all, of a line of a text. If spacing correction is not required anywhere in a line, such as in computer listings, right ragged text, or the last line of a justified paragraph, the CORRECT operator should not be used at all. However, if the line as a whole should have correct measure, but local regions of the line must not participate in the correction calculations, correction must be disabled locally.

In order to disable correction, it suffices to set the imager variable *correctPass* to zero. To enable correction again, *correctPass* should be restored to its former value. So we might use a sequence like:

```
--Example 10.10: temporary disabling of spacing correction--
-- --      --. . . set up CORRECT parameters and set current position--
--0--      {                          --begin CORRECT body--
-- --      --. . . various printing that is subject to correction--
--1--      19 IGET                    --save correctPass on the stack--
--2--      0 19 ISET                  --set correctPass to zero to disable correction--
-- --      --. . . various printing that will not be subject to correction--
--3--      19 ISET                    --restore correctPass to its former value--
-- --      --. . . various printing that is subject to correction--
--4--      } CORRECT
```

As a general rule, correction should be disabled and restored whenever the current position is saved and restored. Example 10.8 saves and restores the current position and should probably also disable and restore the correction information. However, accent character operators are usually designed so as not to allow spacing correction of any sort, so in that example it is not necessary to disable correction. We'll see another use for this disabling correction when a symbol has both a superscript and a subscript.

10.6.8 Superscripts and subscripts

Superscripts and subscripts are placed with relative positioning commands that shift the baseline to a new location and shift it back again when the script is finished. To display the string "D_1", for example, we would use SETYREL to move the current position down for the origin of the subscript and then back up to continue with the rest of the string. Using the conventions for fonts and the page coordinate system established in Example 10.1, we might proceed as follows:

```
--Example 10.11: a subscript--
--0--     0 SETFONT           --10-point times--
--1--     <D> SHOW            --show symbol--
--2--     -40 SETYREL         --move baseline down 4 points--
--3--     3 SETFONT           --8-point times--
--4--     <1> SHOW            --subscript--
--5--     40 SETYREL          --restore baseline--
-- --                         --probably change font back to 10-point times--
```

This code sequence could appear in a CORRECT body or it could appear by itself. Superscripts are handled by using a positive displacement in line 2.

If a symbol has *both* subscripts and superscripts, the solution is a little more complicated. We'll need to remember the current position after the symbol has been printed and restore it for the second subscript. Thus the procedure might be described as: (1) SHOW the symbol; (2) save the current position; (3) use SETYREL to establish the baseline for one of the scripts; (4) SHOW the script; (5) restore the current position; (6) use SETYREL to establish the baseline for the other script; (7) SHOW the script; (8) use SETYREL to return the baseline to its original position. This procedure is illustrated in Example 10.12, which adds a superscript "12.3" to our earlier example:

```
--Example 10.12: subscripts and superscripts on the same character--
--0--     0 SETFONT           --    10-point times--
--1--     <D> SHOW            --(1) show symbol--
--2--     0 IGET 1 IGET       --(2) save current position on the stack--
--3--     -40 SETYREL         --(3) move baseline down 4 points for subscript--
--4--     3 SETFONT           --    8-point times--
--5--     <1> SHOW            --(4) show subscript--
--6--     1 ISET 0 ISET       --(5) restore current position--
--7--     45 SETYREL          --(6) move baseline up 4.5 points for superscript--
--8--     <12.3> SHOW         --(7) show superscript--
--9--     -45 SETYREL         --(8) restore baseline--
```

The procedure outlined above works fine in the absence of correction, as when printing "display math." However, when a symbol with both subscript and superscript appears within a justified line that is adjusted with CORRECT, a slightly different procedure is required. Correction is turned off in the shortest script so as not to mislead CORRECT into thinking the line length can be changed by altering spacing within this script; the line length won't change because we'll restore the current position that we saved, which will cancel any alterations CORRECT might have made. Thus the recommended procedure becomes: (1) SHOW the symbol; (2) save the current position and turn off correction; (3) use SETYREL to establish the baseline for the script with the shortest measure; (4) SHOW the shortest script; (5) restore the current position and correction information; (6) use SETYREL to establish the baseline for the longest script; (7) SHOW the longest script; (8) use SETYREL to return the baseline to its original position. Example 10.13 shows the modifications to Example 10.12 necessary in the presence of correction:

```
--Example 10.13: subscripts and superscripts, adapted to CORRECT operator--
-- 0--    0 SETFONT              --    10-point times--
-- 1--    <D> SHOW               --(1) show symbol--
-- 2--    19 IGET                --(2) save correction state on the stack--
-- 3--    0 19 ISET              --(2) turn off correction--
-- 4--    0 IGET 1 IGET          --(2) save current position on the stack--
-- 5--    -40 SETYREL            --(3) move baseline down 4 points for short script (subscript)--
-- 6--    3 SETFONT              --    8-point times--
-- 7--    <1> SHOW               --(4) show shortest script (subscript)--
-- 8--    1 ISET 0 ISET          --(5) restore current position--
-- 9--    19 ISET                --(5) restore correction state--
--10--    45 SETYREL             --(6) move baseline up 4.5 points for long script (superscript)--
--11--    <12.3> SHOW            --(7) show longest script (superscript)--
--12--    -45 SETYREL            --(8) restore baseline--
```

Another approach would be to turn off correction for both scripts:

```
--Example 10.14: subscripts and superscripts with correction disabled--
-- 0--    0 SETFONT              --    10-point times--
-- 1--    <D> SHOW               --(1) show symbol--
-- 2--    19 IGET                --(x) save correction state on the stack--
-- 3--    0 19 ISET              --(x) turn off correction--
-- 4--    0 IGET 1 IGET          --(2) save current position on the stack--
-- 5--    -40 SETYREL            --(3) move baseline down 4 points for short script (subscript)--
-- 6--    3 SETFONT              --    8-point times--
-- 7--    <1> SHOW               --(4) show shortest script (subscript)--
-- 8--    1 ISET 0 ISET          --(5) restore current position--
-- 9--    45 SETYREL             --(6) move baseline up 4.5 points for long script (superscript)--
--10--    <12.3> SHOW            --(7) show longest script (superscript)--
--11--    -45 SETYREL            --(8) restore baseline--
--12--    19 ISET                --(x) restore correction state--
```

10.6.9 Centering

Text can be centered by computing its measure and positioning it accordingly with SETXY. However, if substantial font approximations are made, the text might no longer appear to be centered. This problem can be solved by using CORRECT, preceding and following the string with generous and equal SPACE, but turning correction off inside the text to be centered. In this way, CORRECT will adjust the SPACE equally so that the text remains centered.

For example, suppose the measure of a string, including its spaces, is 1488 units in the page coordinate system, and that it is to be centered about the point $x=3240$, $y=7200$. We'll put 1000 units of SPACE on each end of the string, as follows:

```
--Example 10.15: centering a string in the presence of font approximations--
--Lines 0 to 6 same as Example 10.1--
-- 7--    {                              --the beginning of page body--
-- 8--    0.000035278 SCALE CONCATT      --set page coordinate system to 1/10 point units--
-- 9--    5 0 SETCORRECTTOLERANCE        --set tolerance to 0.5 points--
--  --                                   --leave correctShrink set to its default value of 1/2--
--10--    3488 0 SETCORRECTMEASURE       --set measure to 1000+1488+1000=3488--
--11--    1496 7200 SETXY                --set starting position x=3240-(3488/2), y=7200--
--12--    {                              --start CORRECT body--
--13--    1000 SPACE                     --first adjustable space--
--14--    19 IGET 0 19 ISET              --turn off correction--
--13--    0 SETFONT                      --sets current font to 10-point times--
--16--    <Introduction to Interpress> SHOW
--17--    19 ISET                        --restore correction--
--18--    1000 SPACE                     --second adjustable space--
--19--    } CORRECT                      --invoke CORRECT--
--20--    }                              --end of the page body--
--21--    END                            --end of the master--
```

10.6.10 Hung text

Sometimes careful thought is required to decide how to use CORRECT. Consider, for example, the following paragraph, with text hung off to the left:

1. The first thing the settlers did was to clear the land of trees and rocks so they could plant crops. This was an arduous task, for they had no . . .

The text on the first line should use CORRECT, since it must remain justified. However, the hung text ("1.") should not be part of the same correction body, or else correction could move the initial "T" and destroy its alignment with the lines below. The proper way to think of this case is as two blocks of text: the hung text, which is left flush, and the paragraph, which is justified. The hung text is typeset with techniques appropriate for left flush text, namely absolute positioning of the starting point of the text, but no CORRECT. The paragraph is typeset just like other justified paragraphs, using CORRECT.

10.7 Summary

This section has presented a number of techniques for achieving high-quality typography in Interpress. The basic facilities of Interpress are quite simple: a master can place a character of arbitrary size and rotation at an arbitrary position on the page. A creator thus has typographic flexibility and complete control.

The situation becomes more complex if we desire to prepare masters that will print acceptably even when a printer must approximate the fonts requested by the master. The principal tools used to retain legibility in these cases are:

- Relative positioning, so that if a character's width differs when printed, other objects related to the character (such as subscripts, underlines, accents, and the succeeding characters on a line) will move to accept the new width. This technique preserves local relationships among objects.
- The CORRECT operator, which enforces a global relationship, namely the measure of a line of text. The operator alters local relationships, but tries to make most of its modifications in inter-word "white space" so as to preserve legibility.

11

Referencing the environment

Although it is possible for an Interpress master to be completely self-contained, most masters make references to important data that lie outside the master itself. These data are contained in an *environment* furnished by the printer. While the most important objects furnished by the environment are fonts and their corresponding metrics, the environment may also contain *decompression operators* that are used in conjunction with scanned images, descriptors for colored ink, and a variety of predefined images such as logos or signatures. It is by inserting into an Interpress master files from the environment that Interpress can print forms of all kinds, such as letterheads, invoice blanks, and so forth. In this way, a master need specify only the filled-in entries on a form and not the form itself.

This section describes conventions for referring to and using the printer's environment. There are two separate mechanisms in Interpress:

- a *naming system* used to extract fonts, decompression operators, and colors from the environment,
- an *encoding mechanism* called *sequenceInsertFile* that is used to insert files from the environment into a master.

11.1 Hierarchical names

Fonts, decompression operators, and colors are named using a compound name, a Vector of Identifiers. We have already seen (Section 3.2.1) how such names are passed to FINDFONT to obtain a font from the environment, e.g., ‹[*xerox, xc82-0-0, times*] FINDFONT›. Similar compound names are used to obtain decompression operators and colors from the environment, topics that we shall not discuss further until Section 15.

The intent of the Interpress naming system is that compound names will be organized in a hierarchical fashion, much like the hierarchical file-naming system of many operating systems (e.g., UNIX†). Interpress does not enforce a hierarchical naming rule, but things will work a lot more smoothly if hierarchical names are used. Moreover, using hierarchical names in no way limits the variety of names that can be used.

†UNIX is a trademark of Bell Laboratories.

In a compound name such as [*a, b, c*], each identifier represents a new level in a name hierarchy. The first identifier, *a*, is in what is called the *Interpress universal registry*. The names in the universal registry are assigned by Xerox in order to be sure that they are unique (see § C). One of the first names assigned was *xerox*. All data that Xerox contributes to an Interpress printer's environment use compound names that begin with the identifier *xerox*, e.g., [*xerox, xc82-0-0, times*].

Other organizations may also have names assigned in the universal registry. The Bank of America, for example, might be assigned the name *bankofamerica*. Data that the Bank of America records in a printer's environment would be named by hierarchical names beginning [*bankofamerica*, . . .]. It's clear that uniqueness of names in the universal registry guarantees that no requests for Bank of America data in a printer's environment will be confused with requests for Xerox data, or with data named by any other client.

When a registry issues a name to a group, it delegates to that group the responsibility for naming at the next level of the hierarchy. For example, the Bank of America will maintain its own registry for names at the second level of the hierarchy; the name at the first level is always *bankofamerica*. This corporate naming body might assign the following names:

bankofamerica standard	for data standard to all Bank of America installations
bankofamerica commercial	for the commercial banking division
bankofamerica investments	for the investments division
bankofamerica branchbanking	for branch banking division

This, of course, simply establishes four new registries for yet another level of names. The process can continue to as many levels in the hierarchy as necessary. Xerox uses levels in its hierarchical names to standardize font names (Section 9.1.1).

The hierarchical name system allows names to be created without any central control. Once Xerox has assigned the name *bankofamerica* in the universal registry, it need have no control whatsoever over further levels in the hierarchy that use names beginning [*bankofamerica*, . . .]. Once the corporate registry of Bank of America has assigned names beginning [*bankofamerica, commercial* . . .] to the commercial banking division, it need not be further concerned with how that division assigns subsequent names. And so on, as deep as necessary.

You might wonder why all this effort is necessary, since an Interpress printer is likely to be owned and operated by a single entity, which could just use local names such as "letterhead." But what happens if a master created by a subsidiary of the company is sent by a computer network to this printer, and refers to "letterhead" as well? The letterhead of the local company, not of the subsidiary, will be printed—the wrong result. If hierarchical names are used instead, the local printer will have [*ussteel, letterhead*] while the master that arrives will request [*ussteel, americanbridge, letterhead*]. In this example, it may happen that the local printer does not have in its environment the letterhead of the subsidiary company, and an error will result. But this behavior is usually preferable to printing the letter on the wrong letterhead!

Because of their virtually limitless ability to expand, hierarchical names can contain a great deal of information. They can easily contain all the information in conventional computer file names, such as user name, directory name, file name, extension, version number, and so on. While not all of these fields make much sense in the Interpress environment, the notion of a version number (or, more precisely, a version identifier) is extremely useful for fonts and forms. Thus, for example, a letterhead form might be named [*ussteel, letterhead, rev1*].

11.2 External references to files

An Interpress master can ask the printer to copy data from another file into the master as it is being executed. The request is carried in the encoded master as a *sequenceInsertFile* encoding-notation (§ 2.5.3). Note that this is *not* an Interpress operator, but rather a feature of the encoding. The idea is that the decoder at the printer, when it encounters a *sequenceInsertFile* in the master, simply redirects its input by ceasing to parse the master and parsing the requested file instead.

The *sequenceInsertFile* encoding-notation takes as its argument a sequence of bytes that is interpreted by the printer as a file name. Interpress imposes no standards on character sets or naming conventions for these file names; the printer may interpret the bytes in an implementation-dependent manner. In this section, we'll assume these names use the ISO 646 character set. Moreover, we'll adopt a convention for showing *sequenceInsertFile* in the written form of a master, namely "++ *filename* ++".

Let's give an example of the way a form might be included in a master. The following master describes a letter that includes the letterhead as a form:

```
--Example 11.1--
-- 0--    BEGIN {                          --beginning of the preamble--
--  --                                     --font 0 is 10-point Times--
-- 1--    [ xerox, xc82-0-0, times ] FINDFONT 10 SCALE MODIFYFONT 0 FSET
--  --                                     --font 1 is 10-point Times Italic--
-- 2--    [ xerox, xc82-0-0, timesitalic ] FINDFONT 10 SCALE MODIFYFONT 1 FSET
-- 3--    }                                --end of the preamble--
-- 4--    {                                --beginning of page body--
-- 5--    ++ xerox standard letterhead ++    --include form --
-- 6--    0.00035278 SCALE CONCATT         --units of 1 point in the page coordinate system--
-- 7--    0 SETFONT                        --use Times 10 point--
-- 8--    108 576 SETXY                    --set the current position to x=1.5 inch, y=8 inch--
-- 9--    <Mr. John Q. Public> SHOW
--10--    108 564 SETXY
--11--    <1456 Ocean Blvd.> SHOW
--  --    --. . . and the rest of the letter --
--12--    }                                --end of the page body--
--13--    END                              --end of the master--
```

Because Interpress places no restrictions on the format of file names used by *sequenceInsertFile*, a printer can adopt file naming conventions that achieve a number of effects:

- File names can be hierarchically structured, using punctuation characters to separate identifiers at different levels of the hierarchy. The name in Example 11.1 is a hierarchical name with identifiers separated by spaces. This naming scheme will allow all the advantages of hierarchical names to accrue to form names as well.
- File names can refer to files that are not actually resident at the printer, but that can be obtained by the printer either by executing some program or by accessing the file through a computer network. If this idea is used in conjunction with hierarchical names, the name *network* might be used at certain levels in the hierarchy to obtain access to the network, e.g., *ussteel network hostname username filename.*
- File names might contain additional information that helps extract only certain parts of a file. For example, if a file is an Interpress master, the preamble and each page body might be separately accessible. The keyword "select-master-part=" might be used to precede a list of numbers indicating which parts to extract, where 0 indicates the preamble, 1 stands for the first page body, and so on. So a file name might be

ussteel standard capitalrequisitionform select-master-part=0,1

which would extract the preamble and first page body and insert them in the requesting master.

- Broadly construed, the *sequenceInsertFile* mechanism is an opportunity to include arbitrary printer-dependent instructions in a master. For example, a file name *do: use purple paper* might be interpreted by the printer as a printing instruction. While this is not the intent of *sequenceInsertFile*, since other mechanisms are provided for communicating printing instructions, Interpress does not prevent this or other unorthodox uses.

Which (if any) of these effects are supported will depend on the implementation of the printer.

11.2.1 The contents of inserted files

It should be clear from Example 11.1 that some strict conventions must be observed by forms and masters to avoid chaos when a master such as this is printed. The problem is that the file requested on line 5 could include an arbitrary sequence of Interpress operators, which could alter the current transformation, change elements of the frame, and so on. The master that asks to insert the file almost certainly depends on some or all of these objects remaining unmodified by the inserted file. In the example above, the letterhead form is clearly not intended to modify the current transformation or frame elements 0 and 1, which contain fonts. We'll see in Section 12 various ways that a master can protect itself against *any* modifications made by the inserted file.

While most forms must not modify objects available to the master, others might be designed specifically to provide certain objects to the master that requests them. For example, some forms might by convention establish in frame element 13 the font that should be used to fill in entries in the form. Or the form might establish a particular coordinate system. Still other forms might expect arguments on the stack. For example, a letterhead that prints the sender's telephone number might expect on the stack a Vector suitable for passing to SHOW to print the telephone number (e.g., <(412) 555-0803>). The letterhead form chooses the position and font for the number, but the master requesting the letterhead provides the number itself.

11.3 Device independence

Any references to the printer's environment limit the device independence of the master because the master may be submitted to a printer that does not have the necessary data in its environment. A master that strives for device independence will limit its requests of the environment to the bare essentials. While it is possible to construct masters that make no requests of the environment whatsoever, these masters may be bulky and awkward; for example, any fonts they use must be defined completely in the master.

In practice, a certain amount of judgement must be used in deciding how to structure a printer's environment and what parts of the environment a master should depend on. While it is difficult to state any general rules that will fit all applications precisely, here are some examples of the reasoning behind various approaches:

- Transient masters can be arbitrarily device-dependent, and may therefore contain arbitrary references to the environment. These are masters that will not be stored permanently and are destined for a single printer or for a set of printers with identical environments. Almost all masters generated for "computer printing" purposes are of this sort.

- Masters that are expected to be spread around an organization can use any features of the environment that are, by management mechanisms, held constant throughout that organization. Thus a memorandum expected to stay within the commercial banking division may use a form named *bankofamerica commercial memoform*, while a memorandum expected to be spread throughout the entire corporation ought to use a form guaranteed to be in the environment of every printer in the corporation, e.g., *bankofamerica memoform*.
- Masters that are to receive wide distribution are likely to print with less difficulty if references to the environment are minimized. Avoid using a memorandum form in the environment, for example, by including in the master Interpress code to construct the memorandum form from more primitive objects. Then if the master is printed on a printer that lacks the form but has the fonts necessary to print the form, it will print properly.

12

Base language II

The Interpress language allows a master to define *composed operators*, which are analogous to the *procedures* found in conventional programming languages. The use of composed operators within masters is similar to that of procedures in computer programs: computations that are repeated often can be defined once in a composed operator and invoked as many times as necessary by calling the operator. The name "composed" comes from the mathematical concept of function composition F∘G and not the typesetter's meaning of "composed." The term *composed operators* distinguishes these operators from *primitive operators*, which are built into Interpress.

This section describes how composed operators are constructed and gives several examples of their use. We have already seen some examples of composed operators: each character in a font is represented as a composed operator, which, when executed, makes an image of the character on the page.

12.1 Constructing and calling composed operators

One way in which composed operators in Interpress differ from the procedures of Pascal is that while a Pascal procedure is declared at compilation time, in the same way that an integer or array would be declared, an Interpress composed operator is an object created at runtime by the execution of a special primitive operator. A composed operator is thus just another data type, and objects of type *Operator* can be left in the stack or stored into the frame just like other values.

A composed operator is constructed using the MAKESIMPLECO operator, which takes a single argument, the body of literals that comprise the composed operator. The result of executing MAKESIMPLECO is a value of type *Operator* left on the stack (§ 2.4.5):

⟨*b:* Body⟩ MAKESIMPLECO → ⟨*o:* Operator⟩

where *o* is a composed operator which has body *b* and initial frame equal to the value of the frame when the MAKESIMPLECO is executed. Note that *bodies* and *operators* are distinct types; MAKESIMPLECO constructs an operator from a body and an initial frame.

The operator that results from a call to MAKESIMPLECO must be saved somewhere so that it can be invoked later. The most common practice is to save the operator in a frame element for use later on.

Once constructed, a composed operator can be called with the DO, DOSAVE, or DOSAVEALL operators. The differences between these operators concern the saving and restoring of imager variables, as explained below in Section 12.1.3. For the time being, we'll use DO to call operators (§ 2.4.5):

⟨*o:* Operator⟩ DO → --the effect on the stack depends on *o*--
where the operator *o* is executed.

Now that we've described operators for defining and calling composed operators, it's time for an example. Example 12.1 constructs and calls a composed operator that draws a square box, one unit on a side.

```
--Example 12.1: defining and calling a composed operator--
--0--     {                          --a bracket that begins the body literal--
--1--     0 0 1 0 MASKVECTOR         --this composed operator will draw a box--
--2--     1 0 1 1 MASKVECTOR
--3--     1 1 0 1 MASKVECTOR
--4--     0 1 0 0 MASKVECTOR
--5--     } MAKESIMPLECO             --close body literal and make composed operator--
--6--     12 FSET                    --save composed operator in frame element 12--
-- --
--7--     12 FGET DO                 --call the composed operator--
```

Lines 0–6 define a composed operator and save it in frame element 12. Line 7 shows the form that a call might take: the composed operator is fetched and then invoked with DO. Calls, of course, can be made wherever and as often as desired. Note that since the composed operator does not change the value of *strokeWidth* or *strokeEnd*, the portion of the master that performs the call is free to change these values between calls.

To help see the similarities between composed operators and procedures of conventional programming languages, here's the Pascal analog of Example 12.1:

```
procedure Box; --a procedure definition--
    begin
    MaskVector(0, 0, 1, 0);
    MaskVector(1, 0, 1, 1);
    MaskVector(1, 1, 0, 1);
    MaskVector(0, 1, 0, 0);
    end;

Box; --a call--
```

Despite this similarity between composed operators and Pascal procedures, there are important differences. We've already mentioned that a composed operator is constructed by *executing* the MAKESIMPLECO operator, while a procedure in a programming language becomes available through a *declaration;* nothing need be executed to define a procedure.

A second difference between Interpress and a conventional programming language concerns *naming.* In an ordinary programming language, there is a mechanism to connect object references to the appropriate object definition by means of identifiers. For example, if you write a Pascal procedure named *Divide* and then elsewhere in the program write *Divide(a,b)*, the Pascal system will recognize that *Divide(a,b)* is a reference to that procedure and will execute the *Divide(a,b)* statement by executing the body of the *Divide* procedure. This linking up of identifiers is so smooth and so universal that programmers are often stymied when they first

encounter a machine language in which they must manually link up object references to the appropriate definitions.

The Interpress base language, like other machine languages, does not provide this automatic name binding or the storage management that it implies. An Interpress program must manage its objects so that they can be referred to by location and not by name. For example, Pascal permits a programmer to define a local variable named *xyz* that will be allocated space on the runtime stack. The Pascal compiler will automatically allocate the correct amount of stack space and will automatically translate a source-code reference to *xyz* into something like "the third variable in the local-variables portion of the stack." The Interpress interpreter performs no such name binding or space management, so the Interpress programmer must refer to storage by position and not by name. Since Interpress masters are never created by hand, but rather are created by a creation program, only the writers of the master creation program need be aware of these issues.

Another common issue in programming languages is the *scope* of identifiers in general, and of procedure names in particular. In Interpress, since a composed operator is a value like any other data value, there are no scope rules. Instead, the rules governing the storage of data values apply. For example, a composed operator defined inside the page body for page 1 cannot be used on page 2, simply because there is no way that data values computed on page 1 can be made available to page 2. (Recall from Section 4.1 that prior to interpreting each page body the stack is cleared and the frame is set to a copy of the frame that resulted from the execution of the preamble.)

12.1.1 Passing arguments to a composed operator

A master can pass arguments to a composed operator by placing them on the stack before calling the composed operator. The composed operator can obtain the arguments by manipulating the stack in any way. It can also return on the stack an arbitrary number of results, which the caller can retrieve. These are the same conventions used by primitive operators in Interpress.

By way of example, let's define a composed operator that we shall name *BuildTransform*, which takes scaling, rotation, and translation parameters and returns a single transformation that accomplishes all three, in the order scale, rotate, translate:

⟨*scale:* Number⟩ ⟨*rotation:* Number⟩ ⟨*tx:* Number⟩ ⟨*ty:* Number⟩ *BuildTransform* → ⟨*t:* Transformation⟩

where *t* = ⟨*scale* SCALE *rotation* ROTATE *tx ty* TRANSLATE CONCAT CONCAT⟩.

The composed operator could be defined as follows:

```
--Example 12.2, define BuildTransform(scale,rotation,tx,ty) as a composed operator--
-- 0--    {                          --a bracket that begins the body literal--
-- 1--    TRANSLATE                  --stack is scale, rotation, T.translate--
-- 2--    EXCH ROTATE EXCH           --stack is scale, T.rotate, T.translate--
-- 3--    3 1 ROLL                   --stack is T.rotate, T.translate, scale--
-- 4--    SCALE                      --stack is T.rotate, T.translate, T.scale--
-- 5--    3 2 ROLL                   --stack is T.scale, T.rotate, T.translate--
-- 6--    CONCAT CONCAT              --stack is T.result--
-- 7--    } MAKESIMPLECO             --close body literal and make composed operator--
-- 8--    34 FSET                    --save composed operator in frame element 34--
--  --
-- 9--    0.0254 90 4.25 5.5 34 FGET DO     --call BuildTransform(0.0254, 90, 4.25, 5.5)--
--10--    CONCATT                    --and concatenate it onto the current transformation--
```

12.1.2 The composed operator's frame

When an Interpress composed operator is invoked, it is given access to a frame that is a *copy* of its creator's frame (not its caller's frame!). It is free to modify that frame using FSET. When a composed operator finishes executing and returns to its caller, its frame is discarded. This means that it is not possible for a composed operator to make any permanent changes to frame data, i.e., changes cannot outlive the execution of the composed operator. A composed operator cannot change the frame of either its caller or its creator, because during its execution the FSET operator will always store into the composed operator's frame copy, and there is no other way to change a frame value.

The initial contents of a composed operator's frame are obtained from a copy of the current frame at the time MAKESIMPLECO was executed to create the composed operator. Thus the definition and execution of a composed operator causes the frame to be copied twice. The first copy is made when MAKESIMPLECO is executed and becomes part of the representation of the composed operator. The second copy is made during the execution of DO, which copies the copy made by MAKESIMPLECO into a scratch frame so that the composed operator is free to modify it during its execution. When the execution of a composed operator terminates, its frame is simply destroyed. The next time DO is called on that composed operator, a fresh copy is made.

These two points are extremely important, for they determine many of the conventions for using composed operators. To repeat:

- When a composed operator is executed, a fresh frame is constructed. The initial contents of this frame are set to the values that were in the frame when MAKESIMPLECO was executed to create the composed operator.
- When a composed operator finishes execution, the values in its frame are discarded.

These two rules are quite pervasive in Interpress. For example, CORRECT executes its body argument by turning it into a composed operator and calling it. As a result, changes to the frame made inside the body argument of CORRECT will not persist after CORRECT exits.

Because the changes made to a frame during execution of a composed operator are not permanent, the operator can use frame elements to store local variables and be guaranteed that this activity will not disrupt the caller. By way of example, consider the following definition of a composed operator *Box*(*width, height*):

```
--Example 12.3, define Box(width, height) as a composed operator--
--0--     {
--1--     0 FSET                                --frame[0]=height
--2--     1 FSET                                --frame[1]=width
--3--     0 0 1 FGET 0 MASKVECTOR               --MaskVector(0,0,width,0)--
--4--     1 FGET 0 1 FGET 0 FGET MASKVECTOR     --MaskVector(width,0,width,height)--
--5--     1 FGET 0 FGET 0 0 FGET MASKVECTOR     --MaskVector(width,height,0,height)--
--6--     0 0 FGET 0 0 MASKVECTOR               --MaskVector(0,height,0,0)--
--7--     } MAKESIMPLECO                        --close body literal and make composed operator--
--8--     32 FSET                               --save composed operator in frame element 32--
-- --
--9--     13 42 32 FGET DO                      --call Box(13, 42)--
```

Because changes to the frame during the execution of a composed operator do not persist, frame elements 0 and 1 will have the same values after the call on line 9 that they had before the call. Actually, this operator makes such simple use of its arguments that saving them in the

frame is not really necessary. However, more complex composed operators would have a difficult time relying on the stack alone for arguments and local variables.

Global variables can be passed *into* a composed operator in a limited way through the initial frame values. For example, if a composed operator is defined *after* fonts have been obtained and saved in frame elements, the composed operator will have access to those fonts as well. Consider the following master, which uses a composed operator to print the page number at the top of each page:

```
--Example 12.4. Similar to Example 6.4--
-- 0--    BEGIN
-- 1--    {                          --beginning of the preamble--
-- --                                --font 0 is 10-point 'LPTA'--
-- 2--    [ xerox, xc82-0-0, lpta ] FINDFONT 100 SCALE MODIFYFONT 0 FSET
-- --                                --font 1 is 10-point 'LPTAbold'--
-- 3--    [ xerox, xc82-0-0, lptabold ] FINDFONT 100 SCALE MODIFYFONT 1 FSET
-- --                                --define a composed operator for heading: Page(<pagenum>)--
-- 4--    {                          --composed operator frame=global frame--
-- 5--    1 SETFONT                  --set font 1--
-- 6--    720 7560 SETXY             --heading at x=1 inch, y=10.5 inch--
-- 7--    <Listing of GP0.PAS at 14:32 on 31 January 1982  Page > SHOW
-- 8--    SHOW                       --show page number, the argument on the stack--
-- 9--    } MAKESIMPLECO 2 FSET      --make composed operator and save as frame[2]--
--10--    }                          --end of the preamble--
--11--    {                          --beginning of the first page body--
--12--    0.000036278 SCALE CONCATT --set the page coordinate system by concatenating onto T--
--13--    <1> 2 FGET DO              --call Page(<1>) to print heading--
--14--    0 SETFONT                  --sets the current font--
--15--    720 7200 SETXY             --top line of listing at x=1 inch, y=10 inch--
--16--    <1  (* GP.PAS -- Simple PASCAL graphics package. *)> SHOW
--17--    720 7080 SETXY             --next line is 12 points below first line--
--18--    <2  const EnterGraphicsMode=29; LeaveGraphicsMode=31;> SHOW
--19--    720 6960 SETXY             --each line is 12 points below previous--
--20--    <3  var xlast,ylast: integer; v: InquiryResponse;> SHOW
--21--    720 6840 SETXY
--22--    <4> SHOW
--23--    720 6720 SETXY
--24--    <5  procedure TransmitCoords(x,y: real);> SHOW
-- --     --more lines of text for the first page would be added here--
--25--    }                          --end of the first page body--
--26--    {                          --beginning of the second page body--
--27--    0.000036278 SCALE CONCATT --set the page coordinate system by concatenating onto T--
--28--    <2> 2 FGET DO              --call Page(<2>) to print heading--
--29--    0 SETFONT                  --sets the current font--
--30--    720 7200 SETXY             --top line of listing at x=1 inch, y=10 inch--
--31--    <51  procedure DrawText(s: string);> SHOW
-- --     --more lines of text for the second page would be added here--
--32--    }                          --end of the second page body--
-- --     --more page bodies for more pages would be added here--
--33--    END                        --end of the master--
```

The composed operator *Page*(*numberString*) is defined in lines 4–9. Note that the <1 SETFONT> in line 5 will reference *frame*[1], thus obtaining the font defined in line 3. The reason is that the initial frame for *Page* will be the value of the frame as of line 9, when the MAKESIMPLECO operator is executed. At this point, *frame*[0] and *frame*[1] have been set up with fonts, on lines 2 and 3. Calls to the *Page* operator are found on lines 13 and 28.

If the *Page* operator were created before the fonts were defined (i.e., if lines 2 and 3 were placed after line 9), the master would not print correctly. The reason is that each time *Page* is invoked, its frame would be set to the frame as of line 9, which would contain all zeroes. As a consequence, the <1 SETFONT> in line 5 would set the current font to the scalar value 0, which causes an error because the current font cannot be a scalar.

12.1.3 Protecting imager variables from a composed operator

Often the caller of a composed operator wants to guarantee that the operator changes only the *output* of the Interpress master (the page image) and not other pieces of state such as frame elements, stack entries, or imager variables. We have already seen that the caller's frame elements are immune in all cases to tampering by a composed operator. The imager variables can be protected as well, depending on the type of call that is used to invoke the composed operator: DO, DOSAVE, or DOSAVEALL. This protection actually takes the form of a save and restore: the composed operator is free to change the imager variables, but they will be restored by DOSAVE or DOSAVEALL to their previous values before control is returned to the body executing the DOSAVE or DOSAVEALL.

DO. If a composed operator is invoked with DO, protection of imager variables is not guaranteed. All changes to imager variables effected by the composed operator will persist after the operator exits.

DOSAVE. If a composed operator is invoked with DOSAVE, only changes made by the composed operator to *persistent* imager variables will persist after the operator exits. The persistent imager variables are the current position and the line measure used by CORRECT (see § 4.2). The other imager variables, such as the current transformation, the current font, and the width of strokes, will be protected.

DOSAVEALL. If a composed operator is invoked with DOSAVEALL, no changes made by the composed operator to imager variables will persist after the operator exits. Thus DOSAVEALL offers complete protection of imager variables.

The DOSAVE operator is particularly useful in imaging applications, because while it protects most of the imager variables, it allows the current position to be changed by an operator. Character operators, for example, are invoked with DOSAVE as part of SHOW's function (§ 4.4.6). This technique protects the current transformation, which is modified as the character operator is invoked to apply the transformations T_{SP} and T_{CS} (Section 6.2.3), but will be restored after the character operator exits. By contrast, the changes made by the character operator to the current position will persist after the operator exits. A character operator invoked in this way is allowed to have the side effects of printing output and of setting a variable to indicate where the next character should be placed, but is prevented from having any other side effects.

In some cases, a master will want to execute a sequence of literals with imager variable protection, but without suffering the overhead of making a composed operator out of them. To get this effect, the master can use the DOSAVESIMPLEBODY operator, defined as equivalent to ⟨MAKESIMPLECO DOSAVE⟩ (§ 2.4.5). For example, consider the *Box*(*width, height*) operator defined in Example 12.3. We can place boxes at different positions on the image by altering the current transformation with a translation transformation, but we don't want the alteration to be permanent. A call on *Box* might therefore look like:

```
--Example 12.5: Example of the use of DOSAVESIMPLEBODY --
--1--     {                              --start body for DOSAVESIMPLEBODY
--2--     121 467 TRANSLATE CONCATT      --alter transformation--
--3--     13 42 32 FGET DO               --call Box(13, 42), operator in frame[32]--
--4--     } DOSAVESIMPLEBODY             --execute lines 2 & 3 but protect transformation--
```

Another example is drawn from Sections 10.6.7ff, in which the current position and/or the correction state must be saved and restored. While the examples we presented there explicitly

saved and restored these variables using the stack, DOSAVE and DOSAVEALL could also have been used. Example 10.13 can be restated as follows:

```
--Example 12.6: Rework of Example 10.13 using DOSAVEALL--
-- 0--    0 SETFONT              --    10-point times--
-- 1--    <D> SHOW               --(1) show symbol--
-- 2--    3 SETFONT              --    8-point times--
-- 3--    {                      --(2) use DOSAVEALL to save correction state & current position--
-- 4--    0 19 ISET              --(2) turn off correction--
-- 5--    -40 SETYREL            --(3) move baseline down 4 points for short script (subscript)--
-- 6--    <1> SHOW               --(4) show shortest script (subscript)--
-- 7--    } MAKESIMPLECO DOSAVEALL --(5) restore correction state & current position--
-- 8--    45 SETYREL             --(6) move baseline up 4.5 points for long script (superscript)--
-- 9--    <12.3> SHOW            --(7) show longest script (superscript)--
--10--    -45 SETYREL            --(8) restore baseline--
```

Note that we've moved the <3 SETFONT> to line 2, outside the body invoked with DOSAVEALL, so that its effect will persist after the call and be available for the SHOW on line 9.

While these examples of protecting parts of a master from making state changes are not particularly compelling because explicit saving and restoring is simple, sometimes we cannot anticipate the parts of the state that must be saved and restored. For example, DOSAVEALL can be used to protect a master against changes made by a file inserted with *sequenceInsertFile*. Line 5 of Example 11.1 could be restated to protect the master as follows:

```
--5--    { ++ xerox standard letterhead ++ } MAKESIMPLECO DOSAVEALL
```

We'll see in Section 16 some additional applications of DOSAVE and DOSAVEALL.

12.1.4 Protecting the stack

While many composed operators use the stack for accepting arguments and returning results, the caller might sometimes wish to insure that the called operator neither pops more arguments from the stack than it should nor places more results on the stack than it should. This is especially important if the composed operator being called is defined outside the master, for example, in a file acquired with *sequenceInsertFile*. In this case, the caller might wish to guard against disagreements about the number of arguments and results that could cause a fatal error in the interpretation of the master.

The MARK, UNMARK, and UNMARK0 operators, described in § 2.4.6, are used to protect the stack. While we won't repeat here definitions of the operators, we'll present a template that can be used to protect the stack:

```
--Example 12.7, template for stack protection with MARK--
--1--    . . .                  --put n arguments on the stack--
--2--    n MARK                 --place a mark on the stack just beyond the arguments--
--3--    . . .                  --call the composed operator using some form of DO--
--4--    m UNMARK               --check to be sure exactly m results are returned--
```

If $m=0$, line 4 can be changed to read simply UNMARK0.

When a master calls composed operators that are constructed within the same master, protecting the stack with marks is not necessary, since the numbers of arguments and results are presumably chosen to agree with the conventions of the operator's definition.

12.1.5 Recommended practice

Although composed operators can be used in many different ways, the following practices will usually suffice:

- In the preamble, define all fonts first, then define composed operators. This technique allows composed operators to obtain fonts from their frames.
- Composed operators refer to the frame to obtain fonts. All other arguments are passed to the composed operator on the stack; of course, the arguments may be stored in the composed operator's frame during its execution.
- If a composed operator needs to call another composed operator, there are two possibilities. If the caller was defined after the called operator, then the called operator's definition will be available in the caller's frame. In this case, the caller can simply use FGET to obtain the called composed operator. If the caller was defined before the called operator, the called operator must be passed on the stack as an argument to the caller, and of course the caller must know to find the operator on the stack.

  As an alternative to storing composed operators in the frame, the entire collection of operators may be stored in a vector. Then, by convention, the vector is passed as the first argument to every composed operator, so that all operators have access to all operators. This technique also allows an operator to call itself recursively.

12.1.6 Uses of composed operators

Composed operators have many applications. Examples are:

- Defining in the preamble a composed operator that is used on each page. Such an operator might be concerned with making images, such as headings, letterheads, or forms. Or it might simply do some calculation, such as setting the initial values of the imager variables on each page.
- Defining a composed operator to make some piece of imagery that will be repeated, possibly in different sizes, orientations, or positions, on one or more pages. Operators of this sort are sometimes called *symbols*, and the images created by calls on them *instances*. Character operators are used in this way. Instances are described in greater detail in Section 14.
- Simplifying the saving and restoring of frame elements or imager variables around a body by constructing a composed operator and using DOSAVE or DOSAVEALL to invoke it. The DOSAVESIMPLEBODY operator is especially convenient for this application.

Composed operators are used implicitly in Interpress in several areas:

- A page body is turned into a composed operator that is then called with DOSAVEALL when the printer needs to create an image of the page (§ 3.1). Because DOSAVEALL is used, a page body cannot make changes to the frame or imager variables that can be detected inside another page body.
- The CORRECT operator and other body operators such as IF, IFELSE, and IFCOPY use the body argument to construct a composed operator that is then called with some form of DO.

The conditional operators, IF, IFELSE, and IFCOPY, are described in the next section.

12.2 Control operators

Interpress provides some operators that affect the flow of control as a master is interpreted. They are intended for use in conjunction with test and arithmetic operators (§§ 2.4.8 and 2.4.9).

⟨*i:* Integer⟩ ⟨*b:* Body⟩ IF → --the effect on the stack depends on *i* and *b* --

where the effect is *b* MAKESIMPLECO DO if $i \neq 0$, and nothing otherwise. Note that neither *i* nor *b* will be on the stack when the body is executed.

This operator allows a body to be executed conditionally. For example, suppose we want to define a composed operator that places a heading on the page, but places the heading at two different places depending on whether the page number is odd or even. The following composed operator *Heading*(*pageNumberString, pageNumber*) will do this:

```
--Example 12.8. Definition of Heading(pageNumberString, pageNumber)--
-- 1--    {
-- 2--    720 7200 SETXY              --set position for even pages (upper left)--
-- 3--    2 MOD 1 EQ                  --compute (pageNumber mod 2)=1 and leave on stack--
-- 4--    {                           --begin conditional body--
-- 5--    5040 7200 SETXY             --set position for odd pages (upper right)--
-- 6--    } IF                        --execute line 5 if pageNumber is odd--
-- 7--    <Page > SHOW                --now put out heading--
-- 8--    SHOW                        --and pageNumberString--
-- 9--    } MAKESIMPLECO 13 FSET      --construct composed operator and save it--
--  --                                --a sample call--
--10--    <104> 104 13 FGET DO        --Heading(<104>, 104)--
```

It is important to realize that the body that is conditionally executed is first converted into a composed operator and therefore will be executed using a *copy* of the frame. Changes made to the frame will not be saved after the body is executed. However, it's still possible to set a frame element conditionally by using the stack to record effects of the conditional evaluation. For example, suppose we want to compute

"if *frame*[2]≠0 **then** *frame*[3] := *frame*[2]"

This can be achieved as follows:

```
--Example 12.9: Conditional copy of a frame element using IF--
--1--     3 FGET                    --get frame[3] on the stack--
--2--     2 FGET 0 EQ NOT           --compute frame[2] ne 0--
--3--     {                         --start conditional body--
--4--     POP                       --remove old frame[3] from the stack--
--5--     2 FGET                    --put frame[2] on the stack--
--6--     } IF                      --execute lines 4 and 5 if frame[2] ne 0--
--7--     3 FSET                    --and save result in frame[3]--
```

Although IF is actually sufficient for all conditional needs, the IFELSE operator makes **if .. then .. else ..** constructs easier:

⟨*i:* Integer⟩ ⟨*b:* Body⟩ IFELSE → --the effect on the stack depends on *i* and *b* --

where the effect is *i b* IF *i* 0 EQ; i.e., it is the same as the effect of IF, followed by pushing 1 if $i=0$ and 0 otherwise. Note that *i* is not on the stack when the body is executed.

The effect of "**if** *i* **then** B_1 **else** B_2" is obtained with ⟨*i* B_1 IFELSE B_2 IF⟩. The effect of "**if** i_1 **then** B_1 **else if** i_2 **then** B_2 **else** B_3" is obtained with ⟨i_1 B_1 IFELSE { i_2 B_2 IFELSE B_3 IF } IF⟩.

12.2.1 The IFCOPY operator

It is often valuable to print from a single master several copies that differ in minor details; for example, each copy might be addressed to a different recipient on the first page. It is important to ensure that these variations do not require the entire master to be reprocessed for each copy. The IFCOPY operator serves this purpose.

⟨*testCopy:* Operator⟩ ⟨*b:* Body⟩ IFCOPY → ⟨⟩

where the body *b* is either executed or not, depending on the outcome of calling the *testCopy* operator. First, *testCopy* is called with two arguments: the copy number (an Integer) and a copy name (an Identifier, obtained from the printing instructions, described in § 3.3), and must return a single Integer. The body *b* is executed unless *testCopy* returns 0.

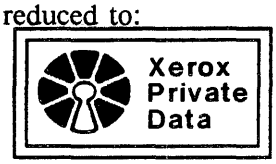
The execution of *testCopy* and *b* are both done with DOSAVEALL to ensure that there are no side effects. Also, neither *testCopy* nor *b* can take any additional arguments off the stack. Thus the net result is that *testCopy* decides whether or not to print the output produced by *b*. A different decision can be made for each copy, but either nothing or the same output is produced each time.

As an example of the use of IFCOPY, the page body below prints a different "To: " field on copies 1 through 4 of the document. The example assumes that the preamble has set up appropriate fonts:

```
--Example 12.10: Using IFCOPY to make variant image copies--
-- 0--    {                          --beginning of a page body--
-- 1--    0.000035278 SCALE CONCATT --set the page coordinate system by concatenating onto T--
-- 2--    0 SETFONT                  --sets the current font--
-- 3--    720 7200 SETXY             --position recipient field at x=1 inch, y=10 inch--
-- 4--    <To: > SHOW                --print 'To: '--
-- 5--    {                          --begin a testCopy operator body--
-- 6--    POP                        --discard copyName--
-- 7--    1 EQ                       --compute copyNumber=1--
-- 8--    } MAKESIMPLECO             --make lines 6 & 7 into an operator--
-- 9--    {                          --begin the body argument to IFCOPY--
--10--    <Robin Richards> SHOW      --copy 1 will read 'To: Robin Richards'--
--11--    } IFCOPY                   --invoke IFCOPY--
--  --
--  --                               --copies 2 through 4 are similar--
--12--    { POP 2 EQ } MAKESIMPLECO
--13--    { <John Calhoun> SHOW } IFCOPY
--  --
--14--    { POP 3 EQ } MAKESIMPLECO
--15--    { <Rachel MacInnes> SHOW } IFCOPY
--  --
--16--    { POP 4 EQ } MAKESIMPLECO
--17--    { <Company archives> SHOW } IFCOPY
--  --
--  --    . . .                      --remainder of page body--
--18--    }                          --end of page body--
```

If a fifth copy of this document is printed, the "To: " herald will be printed, but none of the IFCOPY bodies will be printed. Note that the *copyName* argument to the *testCopy* operators is discarded. The *copyName* argument is provided so that a printer can "name" each copy rather than number it, but for the purposes of this example, the copy number is sufficient.

You might feel that the example above is repetitious, and that we could save space in the master by constructing a composed operator *IfCopyThenShow* that would take two arguments, a copy number and a string to print on that copy. If such an operator were defined in the preamble and saved in frame element 14, the page body above could be reduced to:



```
--Example 12.11: A composed operator to use IFCOPY--
--0--     {                          --beginning of a page body--
--1--     0.000035278 SCALE CONCATT --set the page coordinate system by concatenating onto T--
--2--     0 SETFONT                  --sets the current font--
--3--     720 7200 SETXY             --position recipient field at x=1 inch, y=10 inch--
--4--     <To: > SHOW                --print 'To: '--
--5--     1 <Robin Richards> 14 FGET DO
--6--     2 <John Calhoun> 14 FGET DO
--7--     3 <Rachel MacInnes> 14 FGET DO
--8--     4 <Company archives> 14 FGET DO
-- --     . . .                      --remainder of page body--
--9--     }                          --end of page body--
```

The definition of an appropriate *IfCopyThenShow* operator is:

```
--Example 12.12: definition of IfCopyThenShow(copyNumber, string)--
--0--     {
--1--      0 FSET                    --save string in frame element 0--
--2--      1 FSET                    --save copy number in frame element 1--
--3--      { POP 1 FGET EQ } MAKESIMPLECO    --the testCopy operator--
--4--      { 0 FGET SHOW }           --the body b--
--5--      IFCOPY
--6--     } MAKESIMPLECO 14 FSET     --define composed operator and save in frame[14]--
```

12.3 Summary

This section has dealt with composed operators and the closely related topic of control operators. Several important points about composed operators are worth remembering:

- Each time a composed operator is executed, a fresh frame is constructed. The initial contents of this frame are set to the values in the frame when MAKESIMPLECO was executed to construct the composed operator. Changes made to the frame during execution of a composed operator are discarded when it exits.
- The imager variables can be protected from changes by a composed operator depending on the form of DO used to call the composed operator:

  DO. No protection.

  DOSAVE. All variables protected except the current position and the line measure used by CORRECT.

  DOSAVEALL. All imager variables protected.

It is also important to remember that the encoding (§ 2.5) treats the encoding of an operator that takes a body as argument differently than the encoding of other operator invocations: the token that specifies the operator must *immediately precede* the encoding of the body that is its last argument. This applies to many of the operators described in this section: MAKESIMPLECO, DOSAVESIMPLEBODY, IF, IFELSE, and IFCOPY.

It is time to update Figure 6.8 into Figure 12.1, to include composed operators and their frames.

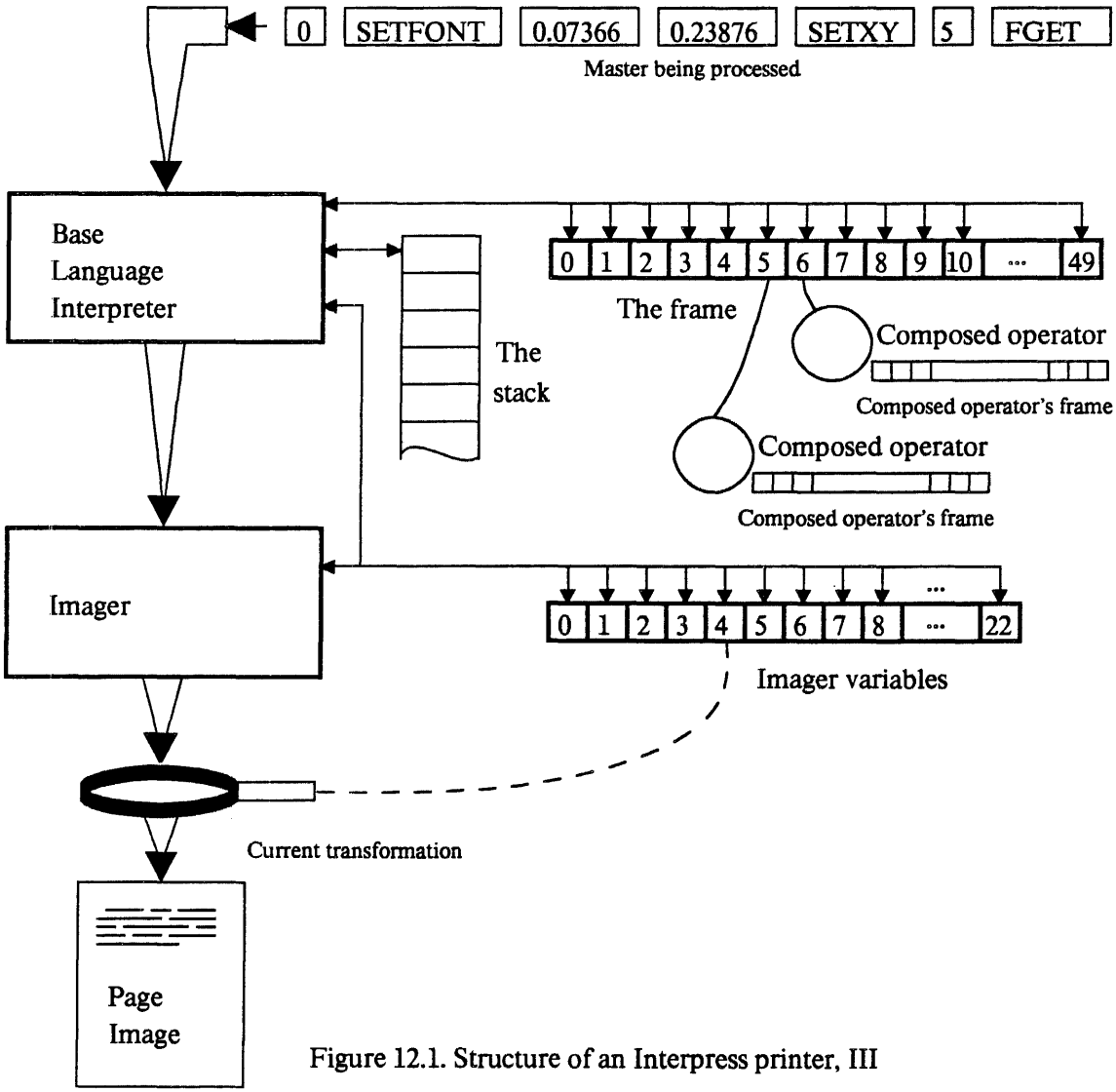


Figure 12.1. Structure of an Interpress printer, III

13

Transformations II

Geometric transformations are at the heart of Interpress imaging operations. We have seen how the choice of master coordinate systems and the size and rotation of characters can be controlled by transformations. So far, the transformation facilities of Interpress have been presented as a set of operators for building primitive transformations (SCALE, ROTATE, TRANSLATE) and two operators for combining transformations (CONCAT, CONCATT).

Underlying the transformation operators is a simple mathematical representation for all kinds of transformations and corresponding rules for transforming a point and for combining transformations. For some people, the mathematical form is simpler to understand than the primitive transformation and combining operators. Moreover, when difficult transformation problems arise, understanding the mathematical underpinnings of transformations can help resolve the problems.

This section explains a mathematical representation of transformations and gives examples of its use. Most printers and many creator programs may choose to represent transformations in the manner described here.

13.1 Primitive transformations

In Section 5.2 we observed that a transformation T_{ft} used to convert from coordinates measured in system f to those measured in system t could be expressed as a linear equation. A particular transformation was illustrated that had the relation:

$$x_t = 2x_f + 7$$
$$y_t = 2y_f + 5$$

All of the primitive transformations have simple relationships of this sort. The list below shows the equations corresponding to each primitive transformation:

$\langle t_x\ t_y$ TRANSLATE$\rangle$

$$x_t = x_f + t_x$$
$$y_t = y_f + t_y$$

⟨s SCALE⟩

$$x_t = sx_f$$
$$y_t = sy_f$$

⟨s_x s_y SCALE2⟩

$$x_t = s_x x_f$$
$$y_t = s_y y_f$$

⟨a ROTATE⟩

$$x_t = cos(a)\ x_f - sin(a)\ y_f$$
$$y_t = sin(a)\ x_f + cos(a)\ y_f$$

While these equations seem simple enough, let's see what happens when we begin combining transformations. Consider the second example shown in Section 6.2.3, where:

T_{CS} = ⟨353 SCALE⟩
T_{SP} = ⟨700 400 TRANSLATE⟩
T_{PI} = ⟨0.00001 SCALE⟩

Converting these transformations into the form shown above, using subscripts on x and y to indicate the coordinate system in which they are measured, we obtain:

$$x_S = 353\ x_C$$
$$y_S = 353\ y_C$$
$$x_P = x_S + 700$$
$$y_P = y_S + 400$$
$$x_I = 0.00001\ x_P$$
$$y_I = 0.00001\ y_P$$

We want to determine a transformation that will convert (x_C, y_C) into (x_I, y_I) by first applying T_{CS}, then T_{SP}, and finally T_{PI}. In Section 6, we learned that such a transformation can be constructed in Interpress by T_{CI} = ⟨T_{CS} T_{SP} T_{PI} CONCAT CONCAT⟩. Using the equation form, the combined transformation can be determined by substituting the first two equations into the second two, and the second two into the third two, to obtain:

$$x_I = 0.00353\ x_C + 0.007$$
$$y_I = 0.00353\ y_C + 0.004$$

We can see that this result is reasonable. For example, the origin (0, 0) of the C coordinate system becomes (0.007, 0.004) in the I system, which can be seen from Figure 6.5 to be correct.

The method of eliminating intermediate coordinate systems by substituting equations will thus lead to compact expressions for a complex transformation. In fact, it turns out that any transformation T_{ft} can always be expressed by two equations of the form:

$$x_t = a\ x_f + b\ y_f + c$$
$$y_t = d\ x_f + e\ y_f + f$$

where a, b, c, d, e, and f are constants determined by the transformations involved. It is clear that each of the primitive transformations can be expressed in this form; it is less clear but nonetheless easily established that an arbitrary combination of the primitive transformations results in equations of this form.

It is fortunate that two linear equations will suffice to represent an arbitrarily complex combination of transformations. This property is what permits Interpress to allow arbitrarily complex transformations in the master and still insure good performance in the application of transformations. At most four multiplications and four additions are required to transform a point in one coordinate system into a point in any other.

While the representation is compact, the rules for combining transformations are not obvious. A matrix notation, explained below, makes the rules more evident.

13.2 The matrix representation of transformations

The most convenient representation of an arbitrary transformation is a 3×3 matrix. Matrices and operations on them are mathematical tools that help deal with linear systems, and indeed the transformation equations we need to represent are linear. We shall represent the general transformation with a matrix of the form:

$$T = \begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

We shall represent a point P whose coordinates are (x, y) as the single-row matrix $[\ x\ y\ 1\ ]$.

The transformation of a point P_f measured in the *from* coordinate system to a point P_t measured in the *to* coordinate system is expressed as the matrix equation $P_t = P_f\ T_{ft}$. This operation is a *matrix multiplication*, which may be visualized by writing out the symbols:

$$[\ x_t\ y_t\ 1\ ] = [\ x_f\ y_f\ 1\ ] \begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

The multiplication requires forming the *inner product* of the single row of P_f with each of the columns of T_{ft}, in order. Thus, x_t is the inner product of $[\ x_f\ y_f\ 1\ ]$ and $[\ a\ b\ c\ ]$, and y_t is the inner product of $[\ x_f\ y_f\ 1\ ]$ and $[\ d\ e\ f\ ]$. The inner product, in turn, is defined as the sum of the products of corresponding terms. Thus the inner product of $[\ x_f\ y_f\ 1\ ]$ and $[\ a\ b\ c\ ]$ is $ax_f + by_f + c$. Note that this inner product is precisely the form we showed for x_t in Section 13.1.

A mnemonic technique may help you to remember how matrix multiplication is done. Take the row-matrix for P_f and turn it on end, first element topmost. Now list it separately next to each column of the transformation matrix, to obtain:

$$\begin{matrix} a\ x_f & d\ x_f & 0\ x_f \\ b\ y_f & e\ y_f & 0\ y_f \\ c\ 1 & f\ 1 & 1\ 1 \end{matrix}$$

Now multiply these pairs together (all nine of them) and sum each column. The resulting three-element row is the matrix product.

13.2.1 Matrix representation of primitive transformations

Using the form outlined above, it is easy to express each of the primitive transformations as a 3×3 matrix. The matrices are:

⟨t_x t_y TRANSLATE⟩

$$T = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ t_x & t_y & 1 \end{matrix}$$

⟨s SCALE⟩

$$T = \begin{matrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{matrix}$$

⟨s_x s_y SCALE2⟩

$$T = \begin{matrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{matrix}$$

⟨a ROTATE⟩

$$T = \begin{matrix} cos(a) & sin(a) & 0 \\ -sin(a) & cos(a) & 0 \\ 0 & 0 & 1 \end{matrix}$$

You may wish to verify that these matrix forms represent the same calculations as the equations in Section 13.1 and that they conform to the intuitive meaning of the transformations they represent.

It is worth noting that the upper left 2×2 matrix elements determine scaling and rotation components of transformations; the first two elements of the last row determine the translation; and the third column is always 0, 0, 1. These properties, it turns out, will remain invariant even as primitive transformations are combined.

13.2.2 Combining transformations

We can now describe the combination, or *concatenation*, of transformations that is performed by CONCAT. Suppose that we have a transformation T_{ab} that transforms from the a system to the b system, i.e., $P_b = P_a T_{ab}$, and a second transformation that transforms from the b system to the c system, i.e., $P_c = P_b T_{bc}$. By substituting the computation of P_b given by the first equation into the second, we obtain $P_c = (P_a T_{ab}) T_{bc}$. Because matrix multiplication is associative, we are allowed to rearrange the parentheses to obtain $P_c = P_a(T_{ab} T_{bc})$. The quantity $(T_{ab} T_{bc})$ is another form of matrix multiplication, this time multiplying two 3×3 matrices together. But we observe an interesting and important result: the final equation is in the same form as the original, that is, $P_c = P_a T_{ac}$, where $T_{ac} = (T_{ab} T_{bc})$. Thus T_{ac} is a new 3×3 matrix that expresses the combined transformation obtained by first applying T_{ab} and then T_{bc}.

The technique for multiplying two 3×3 matrices is similar to the technique for multiplying a single row matrix by a 3×3 matrix: each *row* of the product matrix is obtained by taking the corresponding row of the first matrix and multiplying it by the second matrix using the rule we described above. Thus the following algorithm will compute the matrix $C = AB$:

```
procedure MatrixMultiply(A, B, C: array [1..3, 1..3] of real);
    var row, col, i: integer;
    begin
    for row := 1 to 3 do
        for col := 1 to 3 do
            begin
            C[row, col] := 0;
            for i := 1 to 3 do C[row, col] := C[row, col] + A[row, i]*B[i, col]
            end
    end
```

Note the accumulation of the inner product in the inner-most loop. Also note that each entry in the matrix is denoted by its row and column number, where rows are numbered from 1 at the top to 3 at the bottom and columns from 1 at the left to 3 at the right.

There are three important points to make concerning the concatenation of transformations:

- While matrix multiplication is associative, it is not commutative. That is, the order in which matrices are multiplied is important. By the same token, the order in which transformations are applied is important.
- Arbitrary combinations of primitive transformations always result in the third column of the matrix being 0, 0, 1. The simple form of the third column permits savings in computations.
- Each time two matrices are multiplied, small numeric errors may occur. For example, the concatenation of 45 transformations of ⟨1 ROTATE⟩ will almost certainly not equal precisely the single transformation ⟨45 ROTATE⟩. However, some matrix multiplications need introduce no errors at all, e.g., ⟨90 ROTATE⟩, whose coefficients are either 0, 1, or −1, will not introduce errors into a multiplication. To insure the best precision, masters should avoid concatenating more than 8 transformations together (§ 5.2).

13.2.3 Examples of matrix representations and operations

To return to the example we explored in Section 13.1, let's compute the matrix that expresses the combined transformation T_{CI} = ⟨T_{CS} T_{SP} T_{PI} CONCAT CONCAT⟩. First, let's write the matrices for the individual primitive transformations:

T_{CS} = ⟨353 SCALE⟩

353	0	0
0	353	0
0	0	1

T_{SP} = ⟨700 400 TRANSLATE⟩

1	0	0
0	1	0
700	400	1

T_{PI} = ⟨0.00001 SCALE⟩

0.00001	0	0
0	0.00001	0
0	0	1

Let's start concatenating these transformations together. The transformation T_{CP} = ⟨T_{CS} T_{SP} CONCAT⟩ is obtained by multiplying the matrix representation of T_{CS} by that of T_{SP}:

T_{CP} = ⟨T_{CS} T_{SP} CONCAT⟩

353	0	0
0	353	0
700	400	1

The next step is to compute $T_{CP}T_{PI}$:

T_{CI} = ⟨T_{CS} T_{SP} CONCAT T_{PI} CONCAT⟩

0.00353	0	0
0	0.00353	0
0.007	0.004	1

This result agrees with the result we obtained in Section 13.1 using equations rather than matrices. Given a value for T_{CP}, let's try transforming some points according to that transformation. Consider the point P_C with x_C=0, y_C=0, which is represented as ordinary coordinates (0, 0) or as the row matrix [0 0 1]. So P_I = $P_C T_{CP}$, which we compute to be [0.007 0.004 1]. Thus x_I=0.007 and y_I=0.004, exactly the same results as we obtained with the equation method.

If the order of matrix multiplication is changed, the result will change. For example, you might try computing $T_{PI}T_{SP}T_{CS}$ and comparing the result with the one above.

13.2.4 Concatenating transformations onto the current transformation

There is an important aspect to the order in which transformations are applied that arises when we concatenate transformations onto T, the current transformation. As we've remarked before, the current transformation establishes a current coordinate system. Suppose we name that system A, and we name the current transformation that establishes it T_A. Suppose further that we want to alter the current transformation so that it establishes system C. What's needed is a transformation T_{CA} that we can concatenate onto T_A. If we were to execute ⟨T_{CA} CONCATT⟩, the current transformation would be changed to $T_{CA}T_A$=T_C, just what we want.

It often happens, however, that the incremental transformation T_{CA} is not available as a single transformation, but only as the combination of two separate primitive transformations, T_{CB} and T_{BA}. Because T_{CA}=$T_{CB}T_{BA}$, we could compute T_{CA} with ⟨T_{CB} T_{BA} CONCAT⟩ and then concatenate the result onto T. So the entire computation could be expressed as ⟨T_{CB} T_{BA} CONCAT CONCATT⟩. This will set T to $T_{CB}T_{BA}T_A$=T_C.

Alternatively, suppose we were to concatenate T_{CB} and T_{BA} onto T individually. First, we execute ⟨T_{BA} CONCATT⟩, so that T is set to $T_{BA}T_A$. Then we execute ⟨T_{CB} CONCATT⟩, so that T is set to $T_{CB}T_{BA}T_A$=T_C, the same result as before. The entire computation is therefore ⟨T_{BA} CONCATT T_{CB} CONCATT⟩.

While these two computations are equivalent and equally commendable, there is an important intuitive difference: *the transformations are used in the opposite order* in the two schemes.

Using CONCATT, one envisions building up a transformation from right to left, because CONCATT pre-multiplies the current transformation by the argument to CONCATT:

⟨T_{BA} CONCATT⟩	
	$T_{BA}T_A$
⟨T_{CB} CONCATT⟩	
	$T_{CB}T_{BA}T_A$

By contrast, CONCAT builds transformations left-to-right:

⟨T_{CB}⟩	
	T_{CB}
⟨T_{BA} CONCAT⟩	
	$T_{CB}T_{BA}$
⟨CONCATT⟩	
	$T_{CB}T_{BA}T_A$

The important point is that the various incremental transformations are concatenated together in such a way that when the resulting transformation is pre-multiplied by a point in the appropriate coordinate system, the right effect occurs. It's helpful to write transformations out as a string, e.g.,

$$P_C\,(T_{CB}T_{BA}T_A) = P_A$$

to be sure transformations are being concatenated in the proper order. The subscript notation adds clarity here. A point is subscripted with the name of the coordinate system in which it is measured; a transformation is subscripted with two names—it converts coordinates from the system of the first name to the system of the second name. If this convention is used, then subscripts of adjacent points and transformations in a sequence should agree. In the example above, the subscript C on P_C matches the C on the T_{CB} to the right; the B on T_{CB} matches the B on T_{BA} to the right; and so on. Transformations with a single subscript, such as T_A, denote the current transformation used to establish the coordinate system of the same name, such as A.

13.2.5 Other matrix operations

Matrix multiplication is the only operation necessary to support the geometric transformations we have described so far. There are, however, a few additional concepts that are used in the Standard. They are usually not required to implement a creator program.

Identity. The multiplicative identity matrix, denoted I, has 1's as its diagonal elements and zeroes everywhere else. The identity has the property that for any matrix A, $AI=IA=A$. Note that ⟨1 SCALE⟩, ⟨0 ROTATE⟩, and ⟨0 0 TRANSLATE⟩ all create the identity matrix.

Inverse. The multiplicative inverse of a matrix A is denoted by A^{-1}. It has the property that $A(A^{-1})=(A^{-1})A=I$. Not all matrices have an inverse, e.g., the matrix created by ⟨0 SCALE⟩ has no inverse.

In some cases, inverses can be computed easily. For example, the inverse of ⟨s SCALE⟩ is ⟨1/s SCALE⟩; the inverse of ⟨a ROTATE⟩ is ⟨$-a$ ROTATE⟩; and the inverse of ⟨x y TRANSLATE⟩ is

⟨$-x$ $-y$ TRANSLATE⟩. Moreover, we observe that the inverse of a product of two matrices, $(AB)^{-1}$ is the product of the inverses in reverse order: $(B^{-1})(A^{-1})$.

Proof: $(AB)^{-1}(AB)=I$, by definition of the inverse. Postmultiplying both sides by B^{-1}, we obtain $(AB)^{-1}ABB^{-1}=IB^{-1}$, which simplifies to $(AB)^{-1}A=B^{-1}$. Postmultiplying by A^{-1}, we obtain $(AB)^{-1}AA^{-1}=(B^{-1})(A^{-1})$, which simplifies to $(AB)^{-1}=(B^{-1})(A^{-1})$.

These rules allow us to compute many matrix inverses. If we know the sequence of primitive transformations combined to make the transformation, we invert each primitive transformation according to the rule and invert the product by reversing the order of matrix multiplication. If we are presented with an arbitrary matrix, however, computing an accurate inverse matrix can be a tricky problem.

Transpose. The transpose of a matrix is formed by interchanging its columns with its rows. That is, each element of the matrix, a_{rc}, becomes element a_{cr} of the transpose. Thus, for example, the transpose of a single-row matrix is a single-column matrix.

Additional details on matrix arithmetic can be found in many books on computational linear algebra, such as Forsythe and Moler [6]. Applications to analytic geometry are described in Newman and Sproull and the references it cites [15].

13.2.6 A dual matrix representation

This section is for those whose mathematics schooling has used a different matrix notation for expressing transformations.

The matrix representation presented above is not the only one that can be used to represent the computations required for geometric transformations. Many mathematical treatments of the problem use an approach that is the *dual* of the one above. By dual we mean that the approach above maps directly into the other approach, and for each representation and operation in one scheme there is an exact equivalent in the other scheme. The differences, then, become ones of notational convenience and not of computational power.

The dual notation is:

- Each matrix is represented by the transpose of the ones above. Thus points are represented by single-column matrices and transformations by 3×3 matrices, but with elements transposed.
- The order of matrix multiplication is reversed. Thus we speak of transforming a point by the calculation TP. If transformation T_1 is to be applied first, and then transformation T_2, we must compute $(T_2T_1)P$.

This last example illustrates why we prefer the notation described above: when transformations are to be performed in the order T_1, T_2, and so on, they are written in that same order: $P(T_1T_2\ldots)$. This avoids much confusion.

13.3 The Interpress-to-device transformation

The combination of an arbitrary sequence of transformations into a single matrix is so convenient that Interpress allows the printer to include its device-dependent transformation as part

of the current transformation. This transformation, denoted by T_{ID}, converts coordinates in the Interpress coordinate system (ICS) into a device coordinate system (DCS) suited to the printer (§ 4.3.5). Of course, when a master is created, the value of T_{ID} cannot be anticipated because the master may subsequently be printed on a wide variety of printers with different device coordinate systems.

When execution of a page body is begun, the imager sets the current transformation to T_{ID} (§§ 4.2 and 4.4.5). Thus, coordinates in the master that are expressed in the Interpress coordinate system will be converted by the current transformation into the device's coordinate system.

When the master alters the current transformation, T, it does so by concatenating a new transformation onto the existing current transformation. For example, the transformation T_{PI} = ⟨0.00001 SCALE⟩ is concatenated onto T just after the beginning of the page body in Example 6.2, which is reproduced here for reference:

```
--Example 6.2. Produces the same image as Example 3.2--
--0--     BEGIN { }                --empty preamble--
--1--     {                        --beginning of the page body--
-- --                              --set the page coordinate system by concatenating onto T--
--2--     0.00001 SCALE CONCATT    --TPI = <0.00001 SCALE>--
-- --                              --define a font and save it in frame element 0--
--3--     [ xerox, xc82-0-0, times ] FINDFONT 635 SCALE MODIFYFONT 0 FSET --TCS = <635 SCALE>--
--4--     0 SETFONT                --sets the "current font"--
--5--     7366 23876 SETXY         --sets the "current position"--
-- --                              --TSP = <7366 23876 TRANSLATE>--
--6--     <Interpress> SHOW        --place "Interpress" at current position in current font--
--7--     }                        --end of the page body--
--8--     END                      --end of the master--
```

Before the concatenation on line 2, $T = T_{ID}$. This transformation can be viewed as establishing the Interpress coordinate system. After the concatenation on line 2, $T = T_{PI}T_{ID}$. This transformation can be viewed as establishing the P coordinate system: coordinates are conceptually converted first by T_{PI} from the P system to the I (Interpress) system, and then by T_{ID} to the device's system. Note that the concatenation operation has preserved the Interpress-to-device transformation T_{ID} as part of the combined transformation. Using this kind of concatenation, then, allows the master to derive arbitrary coordinate systems starting from the standard Interpress coordinate system as a reference, but also allows the combined transformation to convert coordinates all the way into the device-dependent coordinate system of the printer.

Interpress does not prevent a master from preparing any transformation it wishes and establishing it as the current transformation. For example, ⟨0 SCALE 4 ISET⟩ will set the current transformation to a particularly useless value. Arbitrary manipulations of the current transformation can make a master device-dependent in the extreme, and are discouraged. The proper practice to insure that the master will print on any device is to use CONCATT to concatenate incremental transformations onto the current transformation, which correctly includes the effect of T_{ID}.

Because Interpress expects the current transformation to convert coordinates all the way to the device coordinate system, several important imager variables are expressed in this coordinate system. The current position and the line measure and tolerance used by CORRECT are expressed in the DCS. The operators provided for setting these variables (SETXY, SETXYREL, SETXREL, SETYREL, SETCORRECTMEASURE, and SETCORRECTTOLERANCE) all transform their arguments using the current transformation before setting the actual imager variables. If a

master changes these variables by other means, it must be sure to account for the current transformation, or at least for the device-dependent transformation T_{ID}. Again, it is safest to use the operators provided.

13.4 Other translation transformations

Interpress provides two special operators, MOVE and TRANS, which work together with the current position to modify the current transformation (§ 4.4.5). MOVE concatenates onto T a translation transformation so that the point (0, 0) will henceforth be mapped to the current position. TRANS does almost the same thing: after the modification to T, the point (0, 0) will be mapped to a point as near to the current position as the printing device can address. TRANS thus moves to the nearest integral point in the device coordinate system.

The objective of these operators is to allow the current position to be used to determine the origin of a symbol to be printed. As we have remarked earlier, character operators set the current position to indicate where the next character in a sequence should be placed. MOVE or TRANS can then be used to modify the current transformation so that the next character operator is executed in a coordinate system that maps the character's origin to the current position. The SHOW operator (§ 4.4.6) uses the TRANS operator to prepare such a transformation. A detailed example of the effect of TRANS can be found in Section 14.4.

13.5 Net transformations

Although Interpress allows characters to be transformed in arbitrary ways as they are placed on the printed image, some printers may not be able to honor this generality. Interpress provides a mechanism for a printer to report to a creator a list of character-to-page transformations that it can handle easily: these are the *easy net transformations* (§ 4.9.3). For example, if a printer can print only 10 and 12 point characters, oriented with a horizontal baseline, it will include in the metric master a list of two easy net transformations associated with this font.

If a creator can make assumptions about which printer will be used to print the document being created, it may wish to limit its character transformations to those in the easy list. Alternatively, the creator may prefer to specify the precise transformations it desires and have the printer make appropriate font approximations and adjustments. In this way, the master adheres more closely to the objective of specifying what the ideal image should look like and assumes the printer will exert its best efforts to produce the ideal image. In any case, the list of easy net transformations is a *hint:* the creator can honor it or ignore it.

Interpress uses the term *net transformation* to describe the transformation from a standard coordinate system to the Interpress coordinate system. The two standard coordinate systems of interest are the character coordinate system and the pixelarray coordinate system (see Section 15.3). A net transformation captures the size and orientation of an object on the final image; any translation component is ignored. Thus, for example, a net transformation of ‹0.0042333 SCALE› describes a 12-point font designed to be read in the normal viewing orientation. This net transformation expresses how the point (0, 1) in the character coordinate system, which is the "body size" of the character, is transformed into the Interpress coordinate system. Scaling by a factor of 0.0042333 will make the body size 0.004233 meters, or 12 points. Similarly, ‹0.00352778 SCALE› describes a 10-point font.

The total transformation applied to a character operator is usually the concatenation of many separate transformations. In Example 6.2, reproduced above, the text "Interpress" is subjected to the following transformations:

T_{CS} = ⟨635 SCALE⟩, specified to MODIFYFONT in line 3
T_{SP} = ⟨7366 23876 TRANSLATE⟩, performed by SHOW on line 6
T_{PI} = ⟨0.00001 SCALE⟩, the page transformation on line 2
T_{ID}, the device-dependent transformation

The second and fourth of these transformations are not included in the net transformation: the second because the net transformation is concerned only with scaling and rotation, and not translation; the fourth because the net transformation records the transformation from the character coordinate system, *C*, to the Interpress coordinate system, *I*, and does not include T_{ID}. So the net transformation is ⟨635 SCALE 0.00001 SCALE CONCAT⟩. Relying either on intuition or on the matrix representation presented in Section 13.2, we see that this transformation can be simplified to ⟨0.00635 SCALE⟩. So if we wanted to be sure a printer could show such a font, we'd look for this transformation in the list of easy net transformations.

Net transformations may also include the effects of rotation. For example, the transformation ⟨0.0035278 SCALE 90 ROTATE CONCAT⟩ describes a 10-point character that runs "up" the page. Example 6.5 uses a font whose net transformation is equal to this one.

When the list of easy net transformations is examined, it's helpful to convert each to a matrix representation and to compare this matrix with the matrix that represents the net transformation the creator would like to use. This is an easy way to detect that ⟨0.00635 SCALE 90 ROTATE CONCAT⟩ and ⟨−270 ROTATE 0.00635 SCALE CONCAT⟩ are equivalent.

13.6 Summary

This section has presented many of the details of the coordinate transformation machinery in Interpress. The key points are:

- Transformations are easily represented as 3×3 matrices. Transforming a point and concatenating transformations are both expressed as matrix multiplications. The matrix notation gives precision both to transformations themselves and to the combination operations.
- The current transformation always includes the effect of a device-dependent transformation T_{ID}, which converts coordinates from the Interpress coordinate system into the device's coordinate system. Masters must ensure that the current transformation always retains the effect of T_{ID}, for example by always concatenating transformations onto the current transformation and never setting the current transformation directly. If the master is written properly, it will be perfectly device-independent in spite of the fact that a device-dependent transformation will always be part of the current transformation.
- Matrix notations provide a convenient way for the creator to test whether a net transformation it desires to apply to a character operator is supported by the printer, i.e., whether the transformation is in the printer's list of easy net transformations.

14

Instancing

The concepts of *symbol* and *instance* are provided in Interpress by composed operators and transformations. A graphical symbol can be defined as a composed operator. When an instance, or copy, of the symbol is to be printed, the current transformation will be applied to all coordinates as the symbol calls imager operators. The properties of the current transformation will thus determine the position, size, and rotation of the instance on the printed page.

The principal use of symbols and instances in Interpress is for printing characters. Each character is defined by a composed operator, called a *character operator.* These operators are invoked, usually by SHOW, with a current transformation that achieves the proper size, orientation, and position of the character.

Instancing can also be used for other purposes. Graphical objects that are repeated often on a page or throughout a document may be treated as instances. A symbol is defined as a composed operator and called with an appropriate current transformation in order to generate each instance. Since SHOW may not be the best operator to effect these calls, other primitives are available as well.

14.1 Defining symbols

A symbol is simply a composed operator that calls imager operators to construct a graphical symbol. The symbol expresses coordinates in a coordinate system of its own choice, sometimes called the *symbol coordinate system.* Figure 14.1 shows a stick figure and an associated coordinate system. Example 14.1 shows how this symbol could be defined as a composed operator. To Interpress, this symbol is simply a composed operator. Its effective use as a symbol depends on the ways in which the master calls the composed operator.

```
--Example 14.1--
--0--     {                                --a symbol is a composed operator--
--1--     0.2 15 ISET                      --set strokeWidth to 0.1 units--
--2--     2 16 ISET                        --set strokeEnd to 2 (round) --
--3--     -2 0 0 4 MASKVECTOR              --left leg--
--4--      2 0 0 4 MASKVECTOR              --right leg--
--5--      0 4 0 8 MASKVECTOR              --torso--
--6--     -2 4 0 6 MASKVECTOR              --left arm--
--7--      2 4 0 6 MASKVECTOR              --right arm--
--8--     -1 9 1 9 MASKVECTOR              --head--
--9--     } MAKESIMPLECO 13 FSET           --make composed operator and save it in frame[13]--
```

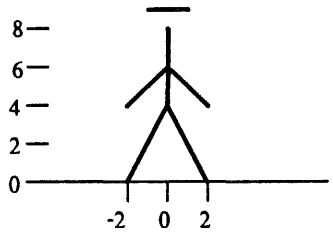


Figure 14.1. A symbol defined in its coordinate system.

14.2 Making instances

In order to place an instance of a symbol on the page, we will need to use a transformation to control the size, rotation, and location of the instance. This transformation is responsible for mapping from the symbol coordinate system to the current coordinate system, the one established by *T*. As we observed in Section 6.2, it's common to establish a page coordinate system that has a convenient origin and units of measurement. The examples in this section will assume a page coordinate system as in Example 6.1, which uses units of 10^{-5} meter.

When we establish the symbol-to-page transformation, it is usually helpful to break the transformation down into two components: (1) a translation, which is used to determine where the origin of the symbol coordinate system will be on the page coordinate system, and (2) a scaling and/or rotation transformation that determines the size and rotation of the instance with respect to its origin.

Suppose, for example, that we want instances of the stick figure in Example 14.1 to be 10 cm. high from toe to head. Since 10 cm. is 10000 units in the page coordinate system, and since the head-to-toe distance is 9 units in the symbol coordinate system, the scale to convert from symbol to page coordinates will be 10000/9.

The following master prints two instances of the symbol, with the same sizes, with origins at $x=5$ cm., $y=8$ cm., and at $x=15$ cm., $y=8$ cm.:

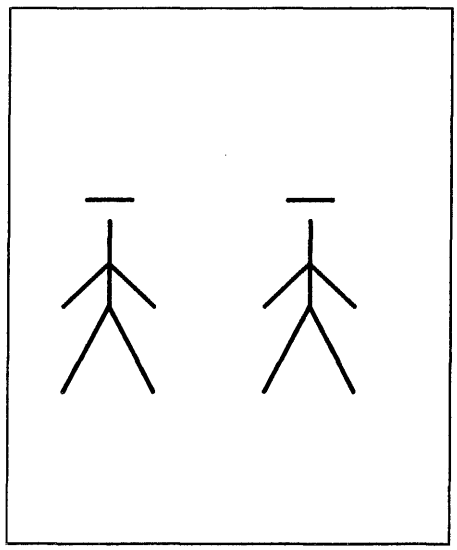


Figure 14.2. Two instances of a symbol.

```
--Example 14.2--
-- 0--    BEGIN {                                --begin preamble--
-- 1--    {                                      --a symbol is a composed operator--
-- 2--    0.2 15 ISET                            --set strokeWidth to 0.1 units--
-- 3--    2 16 ISET                              --set strokeEnd to 2 (round) --
-- 4--    -2 0 0 4 MASKVECTOR                    --left leg--
-- 5--     2 0 0 4 MASKVECTOR                    --right leg--
-- 6--     0 4 0 8 MASKVECTOR                    --torso--
-- 7--    -2 4 0 6 MASKVECTOR                    --left arm--
-- 8--     2 4 0 6 MASKVECTOR                    --right arm--
-- 9--    -1 9 1 9 MASKVECTOR                    --head--
--10--    } MAKESIMPLECO 13 FSET                --make composed operator and save it in frame[13]--
--11--    }                                      --end preamble--
--  --
--12--    { 0.00001 SCALE CONCATT                --page coordinate system in units of 0.00001 meter--
--13--    { 10000/9 SCALE  5000 8000 TRANSLATE CONCAT CONCATT 13 FGET DO } DOSAVESIMPLEBODY
--14--    { 10000/9 SCALE 15000 8000 TRANSLATE CONCAT CONCATT 13 FGET DO } DOSAVESIMPLEBODY
--15--    }                                      --end of page body--
--16--    END
```

Line 13 alters the current transformation so that coordinates in the symbol will be subjected first to a scaling transformation, then a translation, and finally, to the page transformation. Line 13 could equally well be written as ‹{ 10000/9 SCALE 5000 8000 TRANSLATE CONCATT CONCATT 13 FGET DO }›, but could *not* be written as ‹{ 10000/9 SCALE CONCATT 5000 8000 TRANSLATE CONCATT 13 FGET DO }› because this would change the order of application of transformations. However, the following variant would do:

```
--Example 14.3, replacement for line 13 in Example 14.2--
--13--    { 5000 8000 TRANSLATE CONCATT 10000/9 SCALE CONCATT 13 FGET DO } DOSAVESIMPLEBODY
```

You should see clearly how this is equivalent to the original line 13. Understanding the order of application of transformations, and how transformations are concatenated onto *T*, is extremely important.

Note the use of DOSAVESIMPLEBODY to save and restore the current transformation. At the beginning of line 13, the current transformation establishes the page coordinate system. Because of the saving and restoring, this same system will also be in effect after line 13 is executed. If the transformation were not restored, the CONCATT operation on line 13 would have a permanent effect on the current transformation, leaving it in the symbol coordinate system associated with the first instance of the stick figure.

An alternative way to call an instance is to use the current position and MOVE or TRANS to accomplish the translation transformation. Example 14.3 could be modified to read:

```
--Example 14.4, replacement for line 13 in Example 14.2--
--13--    { 5000 8000 SETXY MOVE 10000/9 SCALE CONCATT 13 FGET DO } DOSAVESIMPLEBODY
```

Or, after one minor change, to read:

```
--Example 14.5, replacement for line 13 in Example 14.2--
--13a--   5000 8000 SETXY
--13b--   { MOVE 10000/9 SCALE CONCATT 13 FGET DO } DOSAVESIMPLEBODY
```

If we're going to make lots of instances of the symbol with the same size and orientation, this form has a nice property: line 13b will be the same for each instance because it contains no positioning information. This suggests defining a composed operator to achieve the effect of line 13b. Example 14.6 restates the master in Example 14.2 to use this idea:

```
--Example 14.6, same image as Example 14.2--
-- 0--    BEGIN {                              --begin preamble--
-- 1--    {                                    --a symbol is a composed operator--
-- 2--    0.2 15 ISET                          --set strokeWidth to 0.1 units--
-- 3--    2 16 ISET                            --set strokeEnd to 2 (round) --
-- 4--    -2 0 0 4 MASKVECTOR                  --left leg--
-- 5--     2 0 0 4 MASKVECTOR                  --right leg--
-- 6--     0 4 0 8 MASKVECTOR                  --torso--
-- 7--    -2 4 0 6 MASKVECTOR                  --left arm--
-- 8--     2 4 0 6 MASKVECTOR                  --right arm--
-- 9--    -1 9 1 9 MASKVECTOR                  --head--
--10--    } MAKESIMPLECO 13 FSET               --make composed operator and save it in frame[13]--
--11--    {                                    --another composed operator--
--12--    MOVE 10000/9 SCALE CONCATT 13 FGET DO
--13--    } MAKESIMPLECO 14 FSET               --make composed operator and save it in frame[14]--
--14--    }                                    --end preamble--
--  --
--15--    { 0.00001 SCALE CONCATT              --page coordinate system in units of 0.00001 meter--
--16--    5000 8000 SETXY                      --set current position to x=5 cm, y=8 cm--
--17--    14 FGET DOSAVE                       --print first instance--
--18--    15000 8000 SETXY
--19--    14 FGET DOSAVE                       --print second instance--
--20--    }                                    --end of page body--
--21--    END
```

This example looks a lot like the way character operators work! The composed operator constructed in lines 1 to 10 corresponds to a character operator, defined in the character coordinate system. The composed operator constructed in lines 11 to 13 corresponds roughly to the operator created by MODIFYFONT, which controls the scale and rotation of the instance. The invocation of the operator by first setting the current position to the desired origin of the instance and then calling an appropriate operator (lines 16 and 17) corresponds roughly to the way SETXY and SHOW are used for characters.

The basic Interpress facilities of composed operators and transformations allow a great deal of flexibility in defining and using symbols, including applications we have not illustrated. Symbols may of course be rotated as well as scaled. Symbols can call other symbols, nesting to an arbitrary depth.

14.3 Character operators and instances

Character operators are examples of symbols; in addition to producing an image, these symbols also alter the current position and provide for spacing correction. Example 9.1 presented an example of a symbol for the character '+' which we reproduce here:

```
--Example 9.1--
--0--    {                  --the character operator is defined by the following body literal--
--1--    0.1 15 ISET        --set strokeWidth to 0.1 units--
--2--    0 16 ISET          --set strokeEnd to 0 (square) --
--3--    0.25 0.35 0.65 0.35 MASKVECTOR --horizontal bar of '+'--
--4--    0.45 0.15 0.45 0.55 MASKVECTOR --vertical bar of '+'--
--5--    0.9 0 SETXYREL     --set current position to (0.9,0)--
--6--    CORRECTMASK        --call an operator to correct spacing--
--7--    } MAKESIMPLECO     --the bracket that ends the body literal--
```

The symbol is defined with respect to its own coordinate system (see Figure 5.5). The origin, scale, and directions of coordinate axes are chosen to conform to conventions obeyed by all character operators. FINDFONT will extract from a font library a vector of character operators such as this one.

When MODIFYFONT is applied to a vector of character operators, it creates a new set of operators that first concatenate a transformation onto *T* and then call the character operator. For example, if ⟨*v* 635 SCALE MODIFYFONT⟩ were called on a font vector *v* that included the character operator for '+' given above, the new vector would have, as an operator corresponding to '+', the following:

```
--Example 14.7--
--0--     { 635 SCALE CONCATT co DO } MAKESIMPLECO
```

This operator references *co*, the original character operator. In fact, the entire text of Example 9.1 given above could simply be substituted where *co* appears in Example 14.7 to obtain the composed operator that MODIFYFONT creates. Note the similarities between this operator and lines 11–13 of Example 14.6.

Instances of characters are usually created with SHOW. Before SHOW is called, however, the current position is set to the spot where the origin of the first character should lie. For each character in the vector passed as an argument to SHOW, SHOW executes:

```
--Example 14.8--
--0--     { TRANS mco DO } DOSAVESIMPLEBODY
```

The operator *mco* is a modified character operator, such as shown in Example 14.7, extracted from the *showVec* vector of operators.

14.4 Example: character instancing

This section presents an extended example of a master that prints several characters from a font that it defines. The details of coordinate arithmetic, manipulation of the current transformation, and current position changes are shown.

Our font will have three characters: '+', '−', and 'space,' with indices (or character codes, if you prefer) 0, 1, and 2 respectively.

```
--Example 14.9--
-- 0--    BEGIN {                    --begin preamble--
-- --
-- 1--    {                          --start character operator for '+', index 0--
-- 2--    0.1 15 ISET                --set strokeWidth to 0.1 units--
-- 3--    0 16 ISET                  --set strokeEnd to 0 (square) --
-- 4--    0.25 0.35 0.65 0.35 MASKVECTOR --horizontal bar of '+'--
-- 5--    0.45 0.15 0.45 0.55 MASKVECTOR --vertical bar of '+'--
-- 6--    0.9 0 SETXYREL             --set current position to (0.9,0)--
-- 7--    CORRECTMASK                --call an operator to correct spacing--
-- 8--    } MAKESIMPLECO             --create operator--
-- --
-- 9--    {                          --start character operator for '-', index 1--
--10--    0.1 15 ISET                --set strokeWidth to 0.1 units--
--11--    0 16 ISET                  --set strokeEnd to 0 (square) --
--12--    0.2 0.35 0.6 0.35 MASKVECTOR --horizontal bar of '-'--
--13--    0.8 0 SETXYREL             --set current position to (0.8,0)--
--14--    CORRECTMASK                --call an operator to correct spacing--
--15--    } MAKESIMPLECO             --create operator--
-- --
--16--    {                          --start character operator for ' ', amplifying, index 2--
--17--    0.25 18 IGET MUL 0 SETXYREL       --set current position to (0.25*amplifySpace,0)--
--18--    0.25 18 IGET MUL 0 CORRECTSPACE   --call an operator to correct spacing--
--19--    } MAKESIMPLECO             --create operator--
-- --
--20--    3 MAKEVEC                  --create font vector, similar to that returned by FINDFONT--
```

```
--21--    635 SCALE MODIFYFONT 0 FSET       --make '18 point' font, save in frame[0]--
--22--    }                          --end of preamble--
--  --
--23--    { 0.00001 SCALE CONCATT  --use page coordinate system with units of 0.00001 meter--
--24--    0 SETFONT                --set current font--
--25--    5000 8000 SETXY          --set current position--
--26--    [ 0, 2, 1, 2, 0 ] SHOW   --print '+ - +'--
--27--    }                        --end of page body--
--28--    END                      --end of master--
```

Lines 1 through 20 create on the stack a vector of character operators, analogous to a vector of character operators that would be extracted from the font library by FINDFONT. Line 21 establishes font 0 using the remainder of the font-preparation template introduced in Section 3.2.1. Recall that MODIFYFONT defines a new operator corresponding to each character so as to include the effect of the scaling transformation. For example, the operator stored at index 1 in the vector is:

```
--Example 14.10--
--0--     { 635 SCALE CONCATT      --apply scaling transformation--
--1--     {-                       --start character operator for '-'--
--2--     0.1 15 ISET              --set strokeWidth to 0.1 units--
--3--     0 16 ISET                --set strokeEnd to 0 (square) --
--4--     0.2 0.35 0.6 0.35 MASKVECTOR --horizontal bar of '-'--
--5--     0.8 0 SETXYREL           --set current position to (0.8,0)--
--6--     CORRECTMASK              --call an operator to correct spacing--
--7--     } MAKESIMPLECO           --create character operator--
--8--     DO } MAKESIMPLECO        --create modified character operator--
```

When the page body begins execution on line 23, the current transformation T is set to the Interpress-to-device transformation T_{ID}. For our example, we'll assume a device coordinate system with a resolution of 384 pixels/inch whose origin and coordinate axes are aligned with those of the Interpress coordinate system. Thus T_{ID}=<15118.11 SCALE>, which has the following matrix representation:

$$T = \begin{matrix} 15118.11 & 0 & 0 \\ 0 & 15118.11 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Line 23 concatenates onto T a scaling transformation, so T becomes:

$$T = \begin{matrix} 0.1511811 & 0 & 0 \\ 0 & 0.1511811 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Line 24 simply sets the imager variable *showVec* to contain the font we'll use. Line 25 sets the current position, which involves transforming the point [5000 8000 1] by the current transformation matrix T, given above; thus the current position is set to device coordinates $DCScpx$=755.9055 and $DCScpy$=1209.4488.

On line 26, SHOW swings into action. For the first element in its vector argument, it will execute <{TRANS *mco* DO} DOSAVESIMPLEBODY>, where *mco* is the operator in the vector stored in *showVec* with index 0. First, DOSAVESIMPLEBODY saves all the non-persistent imager variables, including the current transformation we have just illustrated. Then TRANS is called, which rounds the current position to the nearest device coordinate and concatenates an appropriate transformation onto T so that the origin will thereafter be translated to that spot. T becomes:

T=	0.1511811	0	0
	0	0.1511811	0
	756	1209	1

Now the modified character operator is called. The first thing it does (see line 0 of Example 14.10) is concatenate ‹635 SCALE› onto *T*, so *T* becomes:

T=	96	0	0
	0	96	0
	756	1209	1

Now the character operator is executed—in this case, it's the operator for '+' given on lines 1–8 of Example 14.9. The horizontal bar of the character is drawn from (0.25, 0.35) to (0.65, 0.35) in the character coordinate system. These endpoints are transformed by *T* to obtain device coordinates (780, 1242.6) and (818.4, 1242.6), and a stroke whose width is specified by the imager variable *strokeWidth* is drawn on the output device between these endpoints. The vertical bar is handled similarly. Thus the graphics for the character have been drawn on the page image.

Next we encounter line 6, which executes ‹0.9 0 SETXYREL›. This function transforms its arguments using *T*, but by forming the vector [0.9 0 0] rather than the vector [0.9 0 1] to be multiplied by *T*. The reason is that we want *relative* coordinate results, not *absolute* results. We obtain from this multiplication $x=86.4$, $y=0$. These numbers are now added to the current position, in order to achieve the effect of a relative displacement to the current position. The current position becomes *DCScpx*=842.3055 and *DCScpy*=1209.4488. The call to CORRECTMASK does nothing since we're not using CORRECT in this example. So the character operator exits, the modified character operator then exits, and the DOSAVESIMPLEBODY being executed by SHOW is finished. At this point, the imager variables that were saved are restored, and once again the current transformation becomes:

T=	0.1511811	0	0
	0	0.1511811	0
	0	0	1

However, the current position has been permanently changed from *DCScpx*=755.9055 and *DCScpy*=1209.4488 to *DCScpx*=842.3055 and *DCScpy*=1209.4488. This is where the origin of the next character will be placed.

We won't continue the example, but if you want to continue working it yourself and check your work, you may want to verify that the following strokes are drawn, expressed in device coordinates:

(780, 1242.6) to (818.4, 1242.6)	--horizontal bar of +, illustrated above--
(799.2, 1223.4) to (799.2, 1261.8)	--vertical bar of + --
(885.2, 1242.6) to (923.6, 1242.6)	--horizontal bar of – --
(991, 1242.6) to (1029.4, 1242.6)	--horizontal bar of second + --
(1010.2, 1223.4) to (1010.2, 1261.8)	--vertical bar of second + --

After the five-character string has been printed, the current position ends up at *DCScpx*=1053.5055 and *DCScpy*=1209.4488.

The point of this example is to illustrate exactly how instancing occurs. It also shows that character operators and SHOW are simply special cases of a general instancing facility.

This example seems to imply that Interpress will require great quantities of effort just to print simple strings of text. On the contrary, properly designed printer software can avoid the majority of the operations listed above. Note that within a single SHOW command the current transformation matrix changes very little. Even more importantly, a printer will pre-compute how to print each character of a particular size, and so will need only to translate the precomputed form, a very simple calculation. The example shown above, however, illustrates the *effect* that an Interpress printer is required to create.

There is a subtle interplay between TRANS and the current position. TRANS will round the current position when determining a character's translation so that all character instances are simple integral translations on the device's coordinate grid. This step is taken to insure that each instance of a character will appear exactly like all other instances. The small movements introduced by the rounding do not, however, accumulate as errors in the current position because the current position is updated using *relative* positioning commands that are added to the current position and thereby ignore the rounded translation component of the current transformation.

14.5 Example: writing text at an angle

The transformation that places an instance can be used to achieve many geometric effects. For example, it is easy to write lines of text at any angle simply by altering the current transformation suitably before calling the character operators.

The following master shows an example. The font modifications and the page coordinate system are both chosen for a normal page (c.f. Example 10.1).

```
--Example 14.11--
--0--     BEGIN {                          --begin preamble--
-- --                                      --define font 0 to be 10-point times--
--1--     [ xerox, xc82-0-0, times ] FINDFONT 100 SCALE MODIFYFONT 0 FSET
--2--     }                                --end preamble--
--3--     {                                --the beginning of page body--
--4--     0.000035278 SCALE CONCATT        --set page coordinate system to 1/10 point units--
--5--     0 SETFONT                        --sets the "current font" to 10-point times--
--6--     2160 2880 SETXY                  --set current position to x=3 inches, y=4 inches--
--7--     { 45 ROTATE CONCATT <Interpress> SHOW } DOSAVESIMPLEBODY
--8--     }                                --end of the page body--
--9--     END                              --end of the master--
```

The character string printed on line 7 will begin at the point $x=3$ inches, $y=4$ inches and will run up and to the right at a 45 degree angle from the horizontal.

14.6 Summary

The facilities in Interpress for defining composed operators and applying geometric transformations can be used to make instances of graphical symbols on the page image. A special case of instancing is used routinely by the Interpress operator SHOW to place character instances on the page. More general instancing can be invoked explicitly by the master.

15

Graphics

This section explains the Interpress imaging operators for producing graphical images. Interpress provides facilities for generating filled geometric shapes and scanned imagery in addition to the stroke-generation operators already illustrated. Interpress also provides the ability to produce images that are colored or gray.

The notion of a *mask* is central to the graphical primitives. A mask can be visualized as a geometric shape that is laid over the page image in order to determine where to apply *colored ink* to the image. The mask determines exactly where the image will be changed, and the ink, or color, determines how it will be changed. The mask is analogous to an opening in a silk-screen or to the raised area in a woodblock used for printing: it specifies the shape and position of a figure. The color of the figure is determined independently by the ink pressed through the silk screen or applied to the woodblock.

Interpress provides operators for specifying masks and an imager variable that specifies the color of ink to use when a mask is used to change the page image. It is by calling a mask operator that a change is actually made to the page image. All mask operators have the word MASK as part of their name. Each mask operator lays down on the page only a single *graphical primitive.* Complex pages are generated by calling mask operators many times, each time adding a stroke, filled object, or scanned image to the page.

Some Interpress printers may not implement all of the facilities described in this section. Different Interpress subsets (Section 19.1 and § 5.1) provide different facilities.

15.1 Strokes

A stroke is a mask of uniform width used to show a "line" or "rule" on the page. Strokes are used in Interpress to create "line drawings." The term stroke is used rather than line or rule because *line* can be confused with lines of text and *rule* is viewed by some people to be restricted to horizontal and vertical strokes. Almost all of the illustrations in this Introduction are made using only strokes and text.

In its simplest case, a stroke is drawn in a straight line to connect two endpoints, as shown in Figure 15.1. The width of the stroke is specified by an imager variable *strokeWidth*, which the master may set to any desired value. As we'll see a bit later, the treatment of the stroke endpoints is determined by the value of the imager variable *strokeEnd.*

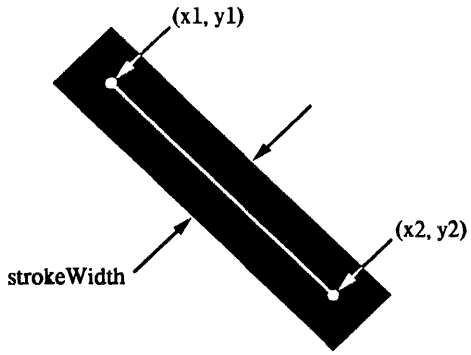


Figure 15.1. A stroke.

A stroke is derived from a *trajectory*. A trajectory is a sequence of points connected by straight line segments. In the case shown in Figure 15.1, the trajectory connects only two points, (x_1, y_1) and (x_2, y_2). A trajectory is an Interpress type that is constructed using the operators MOVETO, LINETO, LINETOX, and LINETOY. The trajectory shown in Figure 15.1 would be constructed by the sequence $\langle x_1\ y_1$ MOVETO $x_2\ y_2$ LINETO$\rangle$, or equivalently, $\langle x_2\ y_2$ MOVETO $x_1\ y_1$ LINETO$\rangle$.

The trajectory-construction operators are described in § 4.8.1. A trajectory is started by calling MOVETO, which takes as arguments the coordinates of the starting point and places on the stack a trajectory value; this represents a trajectory that contains only the starting point. The trajectory on the stack can be extended by calling LINETO, LINETOX, or LINETOY. The LINETO operator takes as arguments the trajectory to be extended and the x and y coordinates of the point to be added to the trajectory. The LINETOX and LINETOY operators are special cases of LINETO that can be used when only the x or y coordinate respectively of the next point in the trajectory differs from the previous point in the trajectory. Thus a horizontal stroke can be constructed with $\langle x_1\ y_1$ MOVETO x_2 LINETOX$\rangle$ since $y_2 = y_1$.

Once a trajectory has been constructed, the MASKSTROKE operator is used to create a mask from the trajectory description and actually alter the page image. MASKSTROKE takes as its only argument the trajectory that will form the center-line of the mask. It is the execution of MASKSTROKE that broadens the trajectory to the width specified by *strokeWidth* and fits endpoints specified by *strokeEnd*. After the stroke is formed from these three ingredients (trajectory, width, endpoints), it is transformed by the current transformation T.

The fact that the mask is first formed and then transformed is important only if the current transformation applies a different scaling to x than it does to y, or involves some other kind of skew transformation. Consider by way of example the stroke illustrated in Figure 15.1, which is shown in the current coordinate system, i.e., the one in which the mask is formed from the trajectory, width, and endpoint specifications. If this figure is transformed by a transformation that magnifies all x coordinates by a factor of two while leaving the y coordinates unchanged, the resulting image will *not* be a rectangular mask. Instead, it will be a parallelogram. This is usually not the effect you want, but there's nothing wrong with using it if you really want it! The same observations apply to the other mask operators as well, all of which apply the current transformation to geometry specified in their arguments.

The convenience operator MASKVECTOR may be used to generate masks for two-point straight-line trajectories. That is, $\langle x_1\ y_1\ x_2\ y_2$ MASKVECTOR$\rangle$ is equivalent to $\langle x_1\ y_1$ MOVETO $x_2\ y_2$ LINETO MASKSTROKE$\rangle$. The MASKVECTOR form is a little shorter and simpler than the MASKSTROKE form.

MASKSTROKE will fit a trajectory with one of three kinds of endpoints, illustrated in Figure 15.2. The *square* endpoint, selected by setting *strokeEnd* to 0, squares off the end of the trajec-

tory after extending it a distance of half its width along the direction in which the trajectory is pointed at the endpoint. The *butt* endpoint, selected by setting *strokeEnd* to 1, simply squares off the trajectory without extending it. Finally, the *round* endpoint, selected by setting *strokeEnd* to 2, places a semicircular cap whose diameter is the same as the stroke width and whose center is located at the trajectory endpoint.

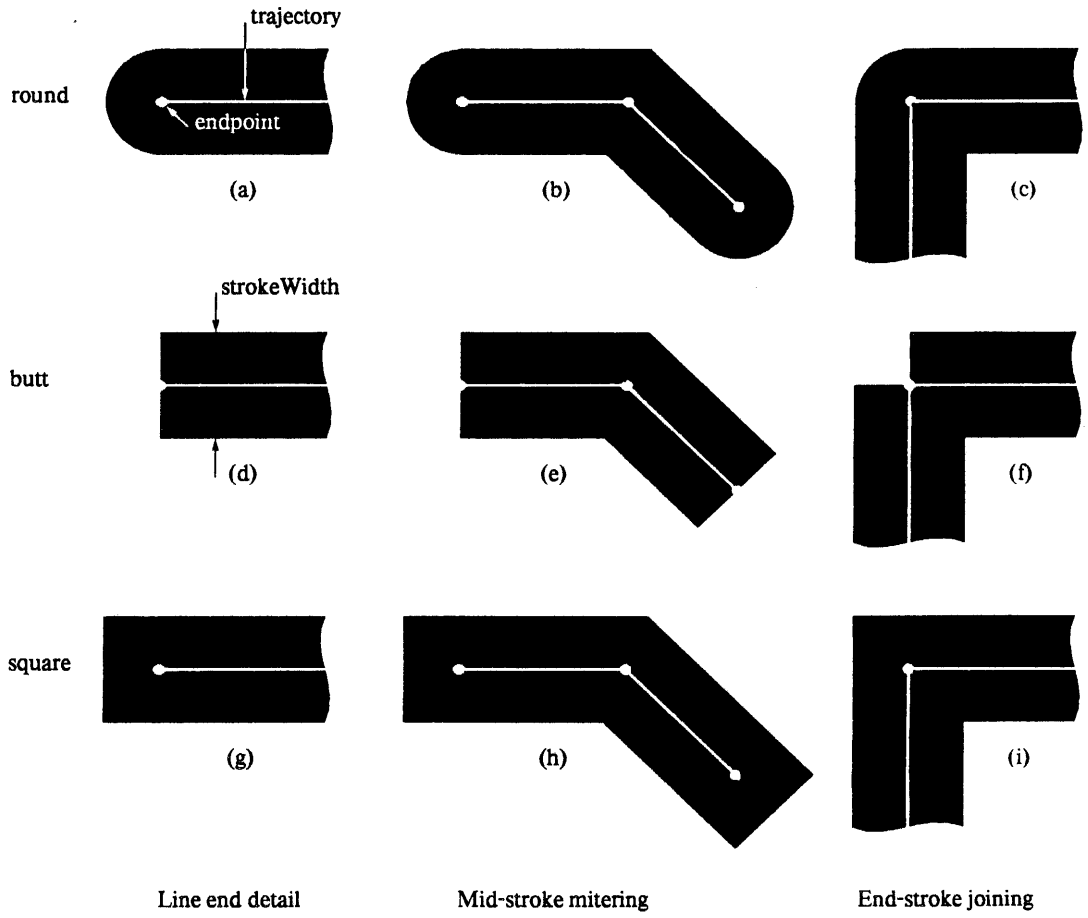


Figure 15.2. Stroke endpoint options (from the Standard)

A trajectory passed to MASKSTROKE may have an arbitrary number of points in it, not just two as shown in Figure 15.1. Regardless of the treatment of the stroke endpoints, all joints between internal segments of the trajectory are *mitered* by extending the sides of the adjacent straight strokes until they meet. This configuration is illustrated in the middle column in Figure 15.2.

As an example of the use of mitering, let's consider drawing the box shown in Figure 3.1. In Example 3.1, this box was drawn using four separate strokes, one for each side of the box. Because *strokeEnd* is set to 0, its initial value established when the page body begins execution, *square* ends are fitted on each stroke. Thus they will join nicely, as illustrated schematically in Figure 15.3. For this application, a joint made with *strokeEnd* set to 1, *butt*, is probably not

desirable (see Figure 15.2f). Exactly the same box can be drawn using a single stroke and relying on mitering to produce three of the four corners. This is illustrated in Example 15.1.

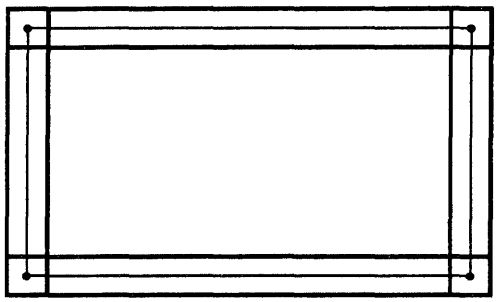


Figure 15.3. A box drawn with four strokes

```
--Example 15.1, same image as Example 3.1--
--0--     BEGIN { }
--1--     { -- assume strokeEnd=0 --
--2--     0.001 15 ISET  --set imager variable 15 (strokeWidth) to 0.001 --
--3--     0.0254 0.2286 MOVETO 0.254 LINETOY 0.1905 LINETOX
--4--       0.2286 LINETOY 0.0254 LINETOX MASKSTROKE
--5--     }
--6--     END
```

If you don't want segments of a trajectory to be mitered, but instead want rounded joints, you must make each segment a separate stroke, fitted with *round* endpoints, as in Figure 15.2c.

It is not wrong to call MASKSTROKE with a trajectory that contains only a single point, the start point. If *strokeEnd* is 0 or 1 (*square* or *butt*) no image will be generated and an appearance error will result, since there is insufficient information to determine the trajectory direction and hence the way to square off the end. However, if *strokeEnd* has the value 2, *round*, a disk of diameter *strokeWidth* will be drawn.

Although Interpress contains no operations for creating curved strokes, curves can be approximated by connecting together a number of short strokes. This approximation is performed by the creator, which puts in the master the calls to generate the appropriate trajectories and strokes. The creator must decide how fine an approximation to use; the decision will depend on a number of factors such as the radius of curvature of the curve, the anticipated viewing distance, and the width of the stroke.

15.2 Filled outlines

Whereas strokes generate masks by broadening from a trajectory center-line, filled outlines generate masks by filling in the region "inside" a closed trajectory. This sort of mask, which is passed as an argument to MASKFILL in order to modify the page image, is used for a wide variety of shapes. For example, filled masks are useful for describing character shapes, which aren't easily modeled with strokes of uniform width.

To make a mask for such a shape, we must first construct an *outline*, an Interpress type constructed with the MAKEOUTLINE operator. An outline is formed from one or more closed trajectories; a closed trajectory is one that closes upon itself by joining the last point on the trajectory to the first point. The definition of MAKEOUTLINE is (§ 4.8.1):

⟨t_1: Trajectory⟩ ⟨t_2: Trajectory⟩ . . . ⟨t_n: Trajectory⟩ ⟨*n:* Integer⟩ MAKEOUTLINE
→ ⟨*o:* Outline⟩

where the trajectories t_1, t_2, . . . t_n together form an outline. Each trajectory will be closed if necessary by linking its last point to its first point.

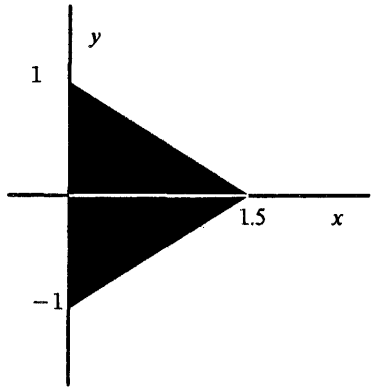


Figure 15.4. A filled outline

Let's consider the example of a solid arrowhead, shown in Figure 15.4. It has a triangular outline that links the points (0, −1), (0, 1) and (1.5, 0). The example uses only a single trajectory, so we can construct the outline of the arrowhead with the sequence ⟨0 −1 MOVETO 1 LINETOY 1.5 0 LINETO 0 −1 LINETO 1 MAKEOUTLINE⟩. Equivalently, we could rely on MAKEOUTLINE to close the trajectory and use ⟨0 −1 MOVETO 1 LINETOY 1.5 0 LINETO 1 MAKEOUTLINE⟩. Also, in this case, it makes no difference whether the triangular shape is traced in a clockwise or counterclockwise order when the trajectory and outline are formed.

Once the outline is prepared, it can be used as a mask by passing it as the only argument to MASKFILL. This operator will first transform the outline according to the current transformation and then alter the portion of the page image covered by the mask. If the current transformation specifies rotation or scaling, the outline will be rotated or scaled accordingly. Example 15.2, below, places an arrowhead such as the one shown in Figure 15.4 pointing upward in the middle of a page. The example uses the instancing techniques explained in Section 14 to transform the coordinate system of Figure 15.4 into the page coordinate system.

```
--Example 15.2--
--0--     BEGIN { }
--1--     {
--2--     0.0254 SCALE CONCATT          --set page coordinate system to inches--
--3--     {
--4--       90 ROTATE 4.25 5 TRANSLATE CONCAT CONCATT
--5--       0 -1 MOVETO 1 LINETOY 1.5 0 LINETO 1 MAKEOUTLINE MASKFILL
--6--     } DOSAVESIMPLEBODY
--7--     }
--8--     END
```

15.2.1 Holes in objects

A filled outline may contain "holes" if the outline is represented properly using several closed trajectories. The basic idea is to use one closed trajectory to specify the outer boundary of the object and a separate closed trajectory to trace out the boundary of each hole in the object. Interpress requires that trajectories that represent hole boundaries be traced in the opposite direction from trajectories that represent the outer boundaries of the object.

Figure 15.5 shows an object with a rectangular outer boundary and a triangular hole boundary. Figures 15.5a and 15.5b use arrows to show that the trajectories are traced in opposite orders. Outlines formed according to these two figures will form rectangular objects with triangular holes.

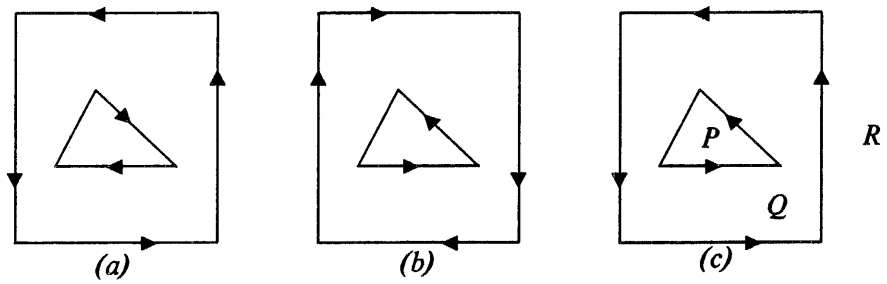


Figure 15.5. Trajectory directions and winding numbers.

If MASKFILL is called with an outline such as the one shown in Figure 15.5c, the result will be a rectangular object with no hole. To understand why this is so, we must explain the rule Interpress uses to decide whether a point is "inside" an outline and hence should be part of the mask. The rule is: a point is inside an outline if and only if its *winding number* is non-zero. Informally, the winding number counts the number of times the outline surrounds the point. More precisely, the winding number can be computed as follows:

1. Set the winding number of the point to 0.
2. For each closed trajectory in the outline that is wound around the point, i.e., that encloses it:
    - 2a. If the trajectory is wound anti-clockwise around the point, add one to the winding number.
    - 2b. If the trajectory is wound clockwise around the point, subtract one from the winding number.

It is clear from this definition that the direction in which a trajectory traces an outline is important in deciding which points will be "inside" the outline.

With this definition of winding number, the treatment of objects in Figure 15.5 can be explained easily. Consider points *P*, *Q*, and *R* shown in Figure 15.5c. Point *R* has a winding number of 0 because neither closed trajectory surrounds it. Hence *R* is not inside the outline. Point *Q* has winding number 1 because the rectangular trajectory is wound anti-clockwise around it but the triangular trajectory is not wound around it at all. Thus point *Q* is inside the outline, because it has non-zero winding number. Point *P* has winding number 2 because both the rectangular and triangular trajectories are wound anti-clockwise about *P*. Hence it too lies inside the outline. If we imagine points *P*, *Q*, and *R* being located analogously on Figures 15.5a and 15.5b as well, we can compute the winding numbers in these cases too:

Case	Point	Winding number of rectangle	Winding number of triangle	Total	Comment
a	*P*	+1	−1	0	not inside
	Q	+1	0	+1	inside
	R	0	0	0	not inside
b	*P*	−1	+1	0	not inside
	Q	−1	0	−1	inside
	R	0	0	0	not inside
c	*P*	+1	+1	+2	inside
	Q	+1	0	+1	inside
	R	0	0	0	not inside

The winding-number rule used by Interpress is not the only acceptable rule for determining whether a point lies inside an outline. Others are: positive winding number, winding number equal to 1, and odd winding number. If an application uses one of these other rules, the creator will have to convert to the Interpress convention as the master is made.

Perhaps the best concrete example of a filled outline is provided by a character operator. Figure 15.6 illustrates a design for an upper case A. Example 15.3 shows a composed operator that will show the character, using one trajectory to represent the outside boundary and one to reqpresent the hole.

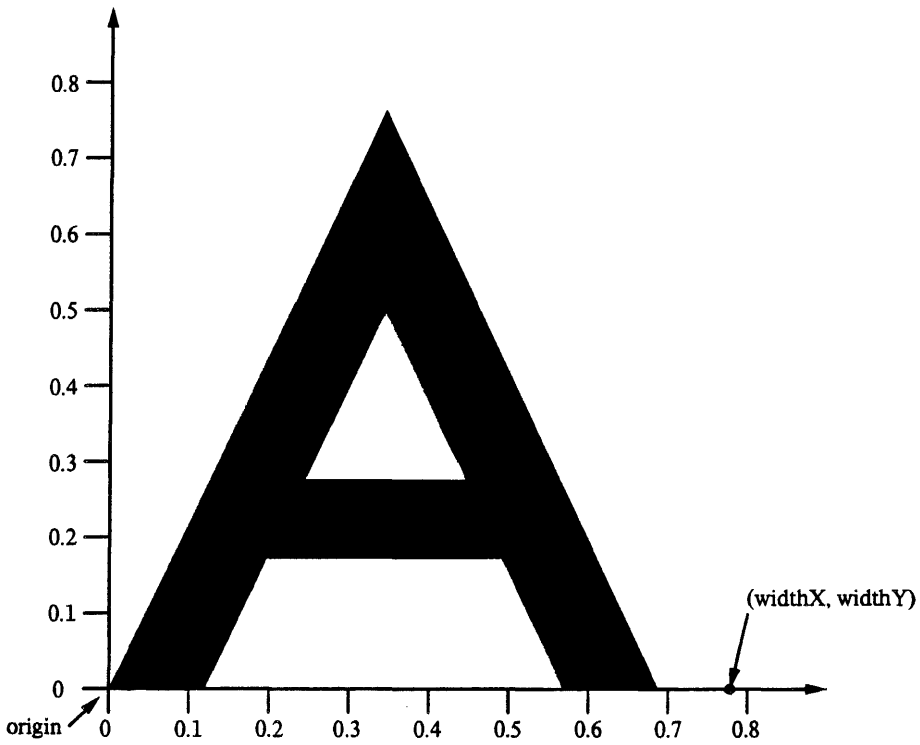


Figure 15.6. An example of a character mask outline

```
--Example 15.3. "A" constructed with a mask having a hole--
-- 0--    {
-- 1--    0 0 MOVETO                  --start first trajectory, outer boundary--
-- 2--    0.120 LINETOX
-- 3--    0.197 0.172 LINETO
-- 4--    0.491 LINETOX
-- 5--    0.568 0 LINETO
-- 6--    0.688 LINETOX
-- 7--    0.344 0.764 LINETO
-- 8--    0  0 LINETO
-- 9--    0.245 0.277 MOVETO          --start second trajectory, hole boundary--
--10--    0.344 0.497 LINETO          --note traversal direction--
--11--    0.443 0.277 LINETO
--12--    2 MAKEOUTLINE MASKFILL      --2 trajectories in the outline--
--13--    0.778 SETXREL               --space over by width--
--14--    CORRECTMASK                 --allow mask spacing correction--
--15--    } MAKESIMPLECO
```

The same shape is specified in Example 15.4 using two masks, one for the left and right arms of the A and one for the horizontal cross bar.

```
--Example 15.4. "A" constructed with two masks--
-- 0--    {
-- 1--    0 0 MOVETO
-- 2--    0.120 LINETOX
-- 3--    0.344 0.497 LINETO
-- 4--    0.568 0 LINETO
-- 5--    0.688 LINETOX
-- 6--    0.344 0.764 LINETO
-- 7--    1 MAKEOUTLINE MASKFILL                  --upside down V --
-- 8--    0.15 0.172 0.4 0.105 MASKRECTANGLE      --horizontal bar--
-- 9--    0.778 SETXREL
--10--    CORRECTMASK
--11--    } MAKESIMPLECO
```

Example 15.4 uses MASKRECTANGLE, which makes a rectangular mask (§ 4.8.2). Its arguments are the *x* and *y* coordinates of the lower-left corner of the rectangle, and the width and height. MASKRECTANGLE is called a *convenience operator* because the same effect can be achieved without its use, but the convenience operator provides a more compact notation.

As we remarked when describing strokes, curved objects must be approximated using outlines consisting of short straight line segments. This is in no way a serious limitation, because the line segments can be arbitrarily short.

15.2.2 Simple outlines

Rectangular and trapezoidal filled outlines occur sufficiently often that Interpress provides special operators to specify them. A solid rectangular mask is generated by ⟨*x y w h* MASKRECTANGLE⟩, where (*x*, *y*) are the coordinates of the lower left-hand corner of the rectangle, *w* is the width, and *h* is the height (§ 4.8.2). Bear in mind that these coordinates will be transformed by the current transformation, thus perhaps rotating or scaling the rectangle—in general, it will appear on the page as a parallelogram. Filled trapezoids may be specified with the MASKTRAPEZOIDX and MASKTRAPEZOIDY operators (§ 4.8.2).

15.3 Scanned images

An Interpress master can describe a mask by providing a raster-scanned image of the mask. Scanned data must be used to represent objects that have no simple geometric form that can

be handled with strokes or outlines. A common use of raster-scanned data occurs when an image has been scanned by a document scanner or facsimile input station and must be represented in an Interpress master. A raster-scanned image may also be created by *halftoning* software in the creator; halftoning converts a continuous-tone photographic image into a pattern of black and white dots that approximates the appearance of the photographic original. Unlike the original, the dot pattern can be printed on a printer that can place only black ink (no gray) on white paper.

The raster data that determines a mask is provided in the form of a two-dimensional array of *sample* values; a sample value of 1 indicates where the mask is to appear and a sample value of 0 indicates where the mask should not appear. The transformation machinery in Interpress is used to map the two-dimensional array onto the page.

15.3.1 Defining a pixel array

To represent a sampled image, Interpress defines a special type, a *pixel array*. As for trajectories and outlines, there is an operator for constructing a pixel array (MAKEPIXELARRAY) and one that uses a pixel array as a mask (MASKPIXEL). The MAKEPIXELARRAY function is quite simple, but appears somewhat complicated because of the provision of transformations. The operator is defined as (§ 4.6):

⟨*xPixels:* Integer⟩ ⟨*yPixels:* Integer⟩ ⟨q_1: Integer⟩ ⟨q_2: Integer⟩ ⟨q_3: Integer⟩
⟨*m:* Transformation⟩ ⟨*samples:* Vector⟩ MAKEPIXELARRAY → ⟨*pa:* PixelArray⟩

where $q_1 = q_2 = q_3 = 1$. The pixel array *pa* is created from *samples* after transformation by *m*.

The construction of the pixel array *pa* created by MAKEPIXELARRAY is best explained in two steps.

1. The first step in the construction of a pixel array places the sample values in a rectangular region of the *pixel array coordinate system*. The samples are used to define a region of width *xPixels* and height *yPixels*, with the lower left corner at the origin. Figure 15.7 shows a pixel array coordinate system with *xPixels*=5 and *yPixels*=8. The coordinate system may be viewed as a grid in which samples are placed; the size of each grid square is one unit.

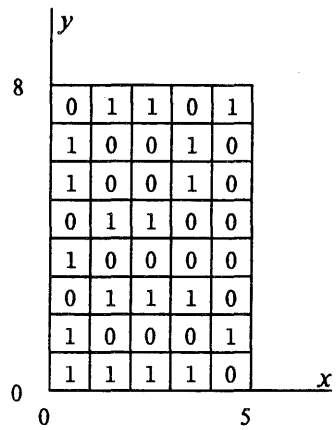


Figure 15.7. A pixel array.

The *samples* vector is used to assign a sample value to each square, or *pixel*, in the grid of the pixel array coordinate system. The first element in the *samples* vector becomes the value of the pixel that touches the origin. The next element is assigned to the pixel immediately above the first one, and so on until the first *yPixels* samples have been assigned. Then the next sample is assigned to the pixel immediately to the right of the first one positioned, and subsequent samples are assigned to pixels of increasing *y* coordinate. In this way, the *samples* vector is used to assign a value to each of the *xPixels*×*yPixels* pixels in the pixel array coordinate system, sweeping out a series of vertical scan-lines at increasing *x* positions; each scan-line is scanned from bottom to top. To test your understanding of this process, you may consult the following example, which builds a pixel array that corresponds to Figure 15.7 (ignore the transformation argument for now):

```
--Example 15.5. Build pixel array for Figure 15.7.--
--0--     5 8                              --xPixels, yPixels--
--1--     1 1 1                            --q's, all =1--
--2--     1 SCALE                          --transformation, for now--
--3--     [ 1, 1, 0, 1, 0, 1, 1, 0,        --first scan line--
--4--       1, 0, 1, 0, 1, 0, 0, 1,
--5--       1, 0, 1, 0, 1, 0, 0, 1,
--6--       1, 0, 1, 0, 0, 1, 1, 0,
--7--       0, 1, 0, 0, 0, 0, 0, 1 ]      --40 elements in samples vector--
--8--     MAKEPIXELARRAY
```

2. The second step in the construction of a pixel array applies a coordinate transformation to the pixel array coordinate system. In a way, this step is similar to the way MODIFYFONT applies a transformation to characters defined in the character coordinate system. In both cases, the transformation is used to convert the standard coordinate system (character or pixel array) into one that is more convenient to use in the master, usually the page coordinate system.

15.3.2 Using a pixel array as a mask

A pixel array is used as a mask by invoking the MASKPIXEL operator. The region defined by each pixel with a sample value of 1 will be part of the mask; pixels with sample values of 0 will not be part of the mask. MASKPIXEL uses the current transformation and the transformation *m* that is part of the pixel array to determine the size, rotation, and position of the mask on the page. The coordinates of a pixel defined in the pixel array coordinate system (Figure 15.7) are transformed first by *m* and then by the current transformation to obtain the position of the pixel on the page.

The following example shows a complete master that will place an image of the pixel array shown in Figure 15.7 on the page. Each pixel will be 1/10 mm square on the page and the lower left corner of the rectangular pixel array will be positioned at $x=10$ cm, $y=13$ cm. You should observe that the final image will resemble a lower-case "g."

```
--Example 15.6. Master that places a pixel array on the page--
-- 0--    BEGIN { }
-- 1--    { 0.00001 SCALE CONCATT          --page coordinate system in units of 0.00001 meter--
-- 2--    5 8                              --xPixels, yPixels--
-- 3--    1 1 1                            --q's, all =1--
-- 4--    10 SCALE 10000 13000 TRANSLATE CONCAT  --transformation--
-- 5--    [ 1, 1, 0, 1, 0, 1, 1, 0,        --first scan line--
-- 6--      1, 0, 1, 0, 1, 0, 0, 1,
-- 7--      1, 0, 1, 0, 1, 0, 0, 1,
-- 8--      1, 0, 1, 0, 0, 1, 1, 0,
-- 9--      0, 1, 0, 0, 0, 0, 0, 1 ]      --40 elements in samples vector--
--10--    MAKEPIXELARRAY MASKPIXEL
--11--    }
--12--    END
```

There are, of course, a great many other ways to achieve the same effect. For example, we could leave the transformation *m* at ⟨1 SCALE⟩ and instead concatentate onto *T* the appropriate scaling and translation transformation before calling MASKPIXEL. The changes to Example 15.6 for this variant are: insert after line 1 the sequence ⟨10 SCALE 10000 13000 TRANSLATE CONCAT CONCATT⟩ and replace line 4 with ⟨1 SCALE⟩. What is important is the combined transformation, ⟨*m T* CONCAT⟩.

Although Interpress masters may specify arbitrary transformations of a pixel array, a printer may not be able to honor all transformations. Some printers will accept only transformations that result in the pixel array being scanned out with scanning directions and resolutions that match those of the printer hardware. More sophisticated printers will tolerate integral scaling, 90-degree rotations, arbitrary scaling, or arbitrary transformations.

The mechanism used to describe a printer's transformation capabilities is the *easy net transformation*, discussed in Section 13.5: the printer lists the easy net transformations that it can apply to pixel arrays. This information is part of a description of a printer's capabilities, but is not available in a standard, computer-readable form. Example 15.6 uses a net transformation of ⟨10 SCALE 0.00001 SCALE CONCAT⟩, which is equivalent to ⟨0.0001 SCALE⟩.

15.3.3 Compressing the sample vector

It is clear from the preceding examples that serious use of pixel arrays will lead to extremely large sample vectors. If an 8½×11 inch page is scanned by a document scanner at 300 pixels/inch, the samples vector will contain 8,415,000 values! A master that encodes such a vector as a *sequenceLargeVector* will be enormous—it must contain at least 8,415,000 bytes. Interpress provides mechanisms to *pack* or *compress* such vectors in order to reduce the size of their representation in the master.

The idea is that an Interpress printer will provide *decompression operators* that take as input a vector of compressed data recorded in the master and produce as output a *samples* vector. These operators have the form:

⟨*v:* Vector⟩ *decompressOperator* → ⟨*samples:* Vector⟩

where *v* is a vector that contains a compressed or packed representation of pixel array samples, as well as any other information that *decompressOperator* may need to unpack or decompress the data.

Interpress does not define these operators as primitive operators because it is impractical to standardize all compression schemes to a sufficient degree. However, a printer will store in its environment a collection of decompression operators that the master can obtain and use. The FINDDECOMPRESSOR operator locates an operator in much the same way FINDFONT locates a font:

⟨*v:* Vector⟩ FINDDECOMPRESSOR → ⟨*o:* Operator⟩

where *v* is a Vector of Identifiers that names the operator to be retrieved from the environment.

The usual way in which these facilities are used is combined with the construction of a pixel array:

```
--0--     xPixels yPixels 1 1 1 m
--1--     [ --vector of compressed data-- ] [ --name-- ] FINDDECOMPRESSOR DO
--2--     MAKEPIXELARRAY
```

To illustrate a concrete case, suppose that the environment contains an operator that will unpack each integer in a vector into 8 one-bit samples starting with the low-order bit of the integer, e.g., 76 is unpacked into 0, 0, 1, 1, 0, 0, 1, 0. This operator will allow us to express the pixel array in Example 15.5 as follows:

```
--Example 15.7, equivalent to Example 15.5--
--0--     5 8                 --xPixels, yPixels--
--1--     1 1 1               --q's, all =1--
--2--     1 SCALE             --transformation--
--3--     [ 107, 149, 149, 101, 130 ] [ unpack8 ] FINDDECOMPRESSOR DO
--4--     MAKEPIXELARRAY
```

The first vector on line 3 can be encoded using a *sequenceLargeVector* with one byte per element ($b=1$) so that every bit in the encoding is meaningful (§ 2.5.3). While this example is not particularly compelling because the pixel array contains only 40 pixels, packing an array reduces the size of its representation by a factor of 8.

Data-compression operators are used in much the same way unpacking operators are, but their internal operation is much more complex than simply unpacking bits. Compression schemes can achieve another factor of 2 to 10 in storage efficiency beyond packing, depending on the kind of scheme used and the contents of the image. Part of the information that must accompany an Interpress printer is a precise description of the unpacking and decompression operators in its environment.

The Xerox decompression and unpacking operators are part of a separate standard, unavailable at the time of this writing.

Interpress printers will take special steps to be sure that decompression operators are executed efficiently. Generally, they will not actually build a vector of samples on the stack, but instead will simply mark the original compressed vector as requiring a certain kind of decompression before it can be used. The decompression will be performed as the image is generated, often by using special-purpose hardware. Thus while the notation of Interpress composed operators is used to describe decompression operators, do not assume that their execution will be unusably slow or that great quantities of storage will be required for the resulting *sample* vector.

15.3.4 Different scanning orders

While it might seem from reading Section 15.3.1 that Interpress pixel arrays must always be scanned with vertical scan-lines moving to the right, such is not the case. The transformation associated with a pixel array by MAKEPIXELARRAY can be used to unscramble any scanning order. For example, suppose scan-lines run vertically, as described above, but run from top to bottom rather than bottom to top. Setting *m* to ⟨1 −1 SCALE2⟩ will transform the pixel array coordinate system so that, when printed on the page, the scan-lines will indeed run from top to bottom. Unfortunately, this transformation will cause the lower left corner of the rectangle containing the scanned data to be at (0, −*yPixels*) rather than (0, 0). But this too can be remedied by setting *m* to ⟨1 −1 SCALE2 0 *yPixels* TRANSLATE CONCAT⟩.

There are, in fact, eight possible scanning orders, and we can derive for each scanning order a transformation to use for *m* that will transform the pixel array coordinate system in such a way that the result is a rectangular region with lower left corner at (0, 0) and upper right corner at some value (*xmax*, *ymax*). To develop all eight transformations, we need a way to describe a scanning order. Two pieces of information are required: the direction in which successive scan-lines are laid down (left-to-right in the standard case described above) and the direction in which successive pixels within a scan line are laid down (bottom-to-top in the case described above). The eight possibilities and corresponding transformations are shown in Table 15.1.

Table 15.1 Transformations for different scanning orders

Scan dir.	Pixel dir.	*xmax*	*ymax*	Transformation *m*
l-r	b-t	*xPixels*	*yPixels*	⟨1 SCALE⟩ (standard order)
l-r	t-b	*xPixels*	*yPixels*	⟨1 −1 SCALE2 0 *yPixels* TRANSLATE CONCAT⟩
r-l	b-t	*xPixels*	*yPixels*	⟨−1 1 SCALE2 *xPixels* 0 TRANSLATE CONCAT⟩
r-l	t-b	*xPixels*	*yPixels*	⟨−1 −1 SCALE2 *xPixels yPixels* TRANSLATE CONCAT⟩
b-t	l-r	*yPixels*	*xPixels*	⟨−90 ROTATE 1 −1 SCALE2 CONCAT⟩
b-t	r-l	*yPixels*	*xPixels*	⟨90 ROTATE *yPixels* 0 TRANSLATE CONCAT⟩
t-b	l-r	*yPixels*	*xPixels*	⟨−90 ROTATE 0 *xPixels* TRANSLATE CONCAT⟩
t-b	r-l	*yPixels*	*xPixels*	⟨90 ROTATE 1 −1 SCALE2 *yPixels xPixels* TRANSLATE CONCAT CONCAT⟩

Note that when thinking about scanning order, it's easier to think of *xPixels* as the number of scan-lines and *yPixels* as the number of pixels in each scan-line. This interpretation is independent of which axis you think of as *x* and which *y*.

Section 4.6 of the Standard describes a convention that can be used for the transformation *m*, namely that after transformation, the pixel array should lie in a rectangular region with the origin at lower left corner when the image appears upright, and the upper right corner of the rectangle at (*xmax*, *ymax*), for *xmax*>0 and *ymax*>0. This is the convention observed in the table.

15.4 Coordinate transformations for masks

The mask operators transform their geometric arguments by the current transformation to determine the position and shape of masks on the page image. Some operators, such as MASKSTROKE, perform geometric computations based on their arguments in order to determine the mask geometry; these computations are performed before the coordinate transformation is done. Thus, for example, MASKSTROKE computes the geometric shape of the stroke from its endpoint coordinates and fits square or round ends on the stroke before the coordinate transformation.

If the current transformation specifies non-uniform scaling, such as different scale factors in the *x* and *y* directions, the image of a mask on the page may seem to be "distorted," that is, its

may call mask operators to place objects on the page and be assured that objects laid down later will have higher priority than objects laid down earlier. In other words, the order of execution of MASK operators determines the priority.

Using this technique, we can see how Figure 15.8 might have been generated. First, *priorityImportant* is set to 1; then the color is set to gray; then MASKFILL is called to generate the parallelogram; then the color is set to black; then MASKVECTOR is called twice to generate the two black strokes; finally, *priorityImportant* can be set back to 0. Because the black strokes are laid down after the parallelogram, they have higher priority. It's a good idea to leave *priorityImportant* set to 0 whenever possible, since the imager must usually work harder to preserve priority than to ignore it.

Let's consider another example, the box shown in Figure 3.1 that is generated by Example 3.1. We can create this box another way, using priority. We first create a solid rectangular mask using black ink. Then we set the ink to "white" and create a second solid rectangular mask that is slightly smaller than the first one, so as to leave a solid outline 1 mm wide. This technique is illustrated in the following master:

```
--Example 15.8, equivalent to Example 3.1. Uses priority and white ink--
--0--     BEGIN { }
--1--     {
--2--     {
--3--     1 5 ISET              --set priorityImportant--
--4--     0.0249 0.2281 0.1661 0.0264 MASKRECTANGLE
--5--     0 SETGRAY
--6--     0.0259 0.2291 0.1641 0.0244 MASKRECTANGLE
--7--     } DOSAVESIMPLEBODY  --restore color and priorityImportant--
--8--     }
--9--     END
```

This technique is not generally recommended, since it's almost certainly slower to create two large filled rectangles than four strokes, as in Example 3.1. A more interesting and appropriate use of "white" ink prints the lettering in Figure 15.2a.

When priority is used, it's important to be sure that *all* objects whose priority is important are generated when *priorityImportant* has a non-zero value. A change to the page image induced by a mask operator is said to be *ordered* if the mask operator is executed when *priorityImportant* is not zero, and *unordered* if it is zero. Interpress preserves the priority order of all ordered masks, but may alter the priority among unordered masks or between any unordered mask and an ordered mask. To illustrate the use of this rule, consider Example 3.5, which extended Example 3.1 to print the word "Interpress" in the middle of the box. Were we using the technique in Example 15.8 to generate the box, the analog of Example 3.5 would be:

```
--Example 15.9, equivalent to Example 3.5--
-- 0--    BEGIN { [ xerox, xc82-0-0, times ] FINDFONT 0.00635 SCALE MODIFYFONT 0 FSET }
-- 1--    {
-- 2--    {
-- 3--    1 5 ISET              --set priorityImportant--
-- 4--    0.0249 0.2281 0.1661 0.0264 MASKRECTANGLE
-- 5--    0 SETGRAY
-- 6--    0.0259 0.2291 0.1641 0.0244 MASKRECTANGLE
-- 7--    1 SETGRAY
-- 8--    0 SETFONT 0.07366 0.23876 SETXY <Interpress> SHOW
-- 9--    } DOSAVESIMPLEBODY  --restore color and priorityImportant--
--10--    }
--11--    END
```

Note that the text must take priority over the "white" filled outline. If lines 7 and 8 were moved to just before line 5, the text would have lower priority than the white rectangle and would not appear. If lines 7 and 8 were moved to after line 9 or before line 2, the calls to mask operators that generate the string would occur when *priorityImportant* is 0. As a consequence, the text would be unordered with respect to the filled outlines. In this case, the imager might not produce the correct image.

15.7 Summary

Interpress provides a small but complete set of facilities for producing graphical images:

- Strokes, for drawing "lines" of various widths with square or rounded ends.
- Filled outlines for making solid figures of various sorts, such as characters, bar charts, and pie charts.
- Scanned images, for reproducing images scanned on a raster input scanner or for presenting halftoned renditions of photographic images.
- Colors, including shades of gray and any other colors that the printer can achieve.
- Priority, to resolve the ambiguity when objects of different colors overlap.

All of the illustrations in this Introduction are produced using these functions.

There are, in fact, eight possible scanning orders, and we can derive for each scanning order a transformation to use for *m* that will transform the pixel array coordinate system in such a way that the result is a rectangular region with lower left corner at (0, 0) and upper right corner at some value (*xmax*, *ymax*). To develop all eight transformations, we need a way to describe a scanning order. Two pieces of information are required: the direction in which successive scan-lines are laid down (left-to-right in the standard case described above) and the direction in which successive pixels within a scan line are laid down (bottom-to-top in the case described above). The eight possibilities and corresponding transformations are shown in Table 15.1.

Table 15.1 Transformations for different scanning orders

Scan dir.	Pixel dir.	*xmax*	*ymax*	Transformation *m*
l-r	b-t	*xPixels*	*yPixels*	⟨1 SCALE⟩ (standard order)
l-r	t-b	*xPixels*	*yPixels*	⟨1 −1 SCALE2 0 *yPixels* TRANSLATE CONCAT⟩
r-l	b-t	*xPixels*	*yPixels*	⟨−1 1 SCALE2 *xPixels* 0 TRANSLATE CONCAT⟩
r-l	t-b	*xPixels*	*yPixels*	⟨−1 −1 SCALE2 *xPixels yPixels* TRANSLATE CONCAT⟩
b-t	l-r	*yPixels*	*xPixels*	⟨−90 ROTATE 1 −1 SCALE2 CONCAT⟩
b-t	r-l	*yPixels*	*xPixels*	⟨90 ROTATE *yPixels* 0 TRANSLATE CONCAT⟩
t-b	l-r	*yPixels*	*xPixels*	⟨−90 ROTATE 0 *xPixels* TRANSLATE CONCAT⟩
t-b	r-l	*yPixels*	*xPixels*	⟨90 ROTATE 1 −1 SCALE2 *yPixels xPixels* TRANSLATE CONCAT CONCAT⟩

Note that when thinking about scanning order, it's easier to think of *xPixels* as the number of scan-lines and *yPixels* as the number of pixels in each scan-line. This interpretation is independent of which axis you think of as *x* and which *y*.

Section 4.6 of the Standard describes a convention that can be used for the transformation *m*, namely that after transformation, the pixel array should lie in a rectangular region with the origin at lower left corner when the image appears upright, and the upper right corner of the rectangle at (*xmax*, *ymax*), for *xmax*>0 and *ymax*>0. This is the convention observed in the table.

15.4 Coordinate transformations for masks

The mask operators transform their geometric arguments by the current transformation to determine the position and shape of masks on the page image. Some operators, such as MASKSTROKE, perform geometric computations based on their arguments in order to determine the mask geometry; these computations are performed before the coordinate transformation is done. Thus, for example, MASKSTROKE computes the geometric shape of the stroke from its endpoint coordinates and fits square or round ends on the stroke before the coordinate transformation.

If the current transformation specifies non-uniform scaling, such as different scale factors in the *x* and *y* directions, the image of a mask on the page may seem to be "distorted," that is, its

shape will not be the same on the page as in the master coordinate system. For example, a rectangle may become a skew parallelogram or a circular outline may become ellipsoidal. The ellipse results because the non-uniform scaling grows one axis of the circle more than the other. While there is nothing wrong with non-uniform scaling transformations, their use can be confusing. For example, a stroke with a "square" end will not be rectangular. For this reason, non-uniform scaling is not recommended.

15.5 Color

Interpress can describe colored images. When a mask operator is called, the setting of the *current color* determines the color that will be used to print the graphical object defined by the mask. Interpress colors include all shades of gray, including white, as well as a wider gamut of colors on those printers that have inks or toners other than black.

Interpress sets the current color to any shade of gray with the SETGRAY operator (§ 4.7):

⟨*f:* Number⟩ SETGRAY → ⟨⟩

where the current color is set to a shade of gray. The Number *f*, $0 \leq f \leq 1$, specifies the fraction of incident light that will be absorbed by the ink that is deposited on the page. The current color is held in the imager variable *color*, index 13.

Thus black ink is specified with ⟨1 SETGRAY⟩, which is the default established at the beginning of execution of each page body. Executing ⟨0.5 SETGRAY⟩ obtains an intermediate shade of gray. The setting ⟨0 SETGRAY⟩ specifies white. (It may seem senseless to use white color on white paper, but we'll see below in Section 15.6 how white color can be useful.)

A second way to set the color is provided by the FINDCOLOR operator, useful for those printers that can obtain colors other than grays. FINDCOLOR takes as its argument the name of a color, which is used to find an appropriate Color in the printer's environment:

⟨*v:* Vector⟩ FINDCOLOR → ⟨*col:* Color⟩

where *v* is a Vector of Identifiers that names a color. The color returned may be used to set the *color* variable. For example, ⟨[*xerox, highlight*] FINDCOLOR *color* ISET⟩ might set the current color to a "highlight" color.

Some printers may have no colors available to FINDCOLOR, others may supply a single highlight color. Full-color printers may offer a wide range of colors, obtained using naming system that might include such names as "blue-green" or "light brown."

Interpress also provides a third way to set color by using a pixel array to define a black-and-white pattern throughout the page. The interested reader is urged to consult § 4.7 for a detailed description of these facilities; we shall not cover them here.

Figure 15.8 shows an example of color in use. The parallelogram labeled "Color" is created by setting the color with ⟨0.5 SETGRAY⟩ and then using a filled outline mask to define the parallelogram's shape. The "b" on the page image in that figure is printed using the same color setting and a filled outline mask that defines the character's shape.

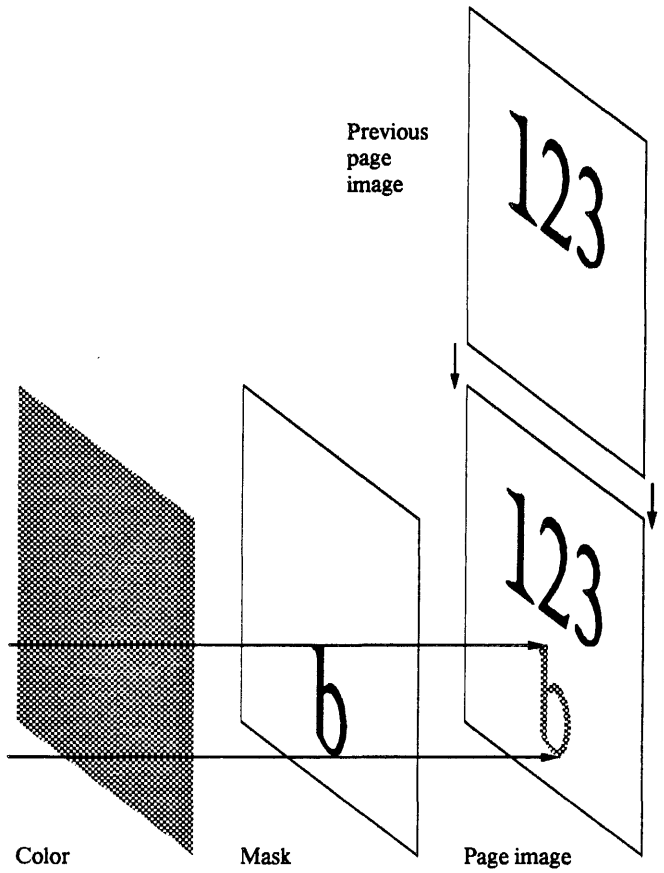


Figure 15.8. Interpress imaging model (from the Standard).

15.6 Priority

When more than one color is used on a page and masks of two different colors overlap, there is a possible ambiguity about which mask or color will be visible. This problem is illustrated well by Figure 15.8: two horizontal black strokes appear to pass in front of the "color" parallelogram at the left of the figure. Here two black objects overlap a gray one. Which objects should be visible? Should the black lines appear to pass in front of the parallelogram or behind? The decision, of course, is up to the artist who prepares the illustration. But how is the decision reflected in the Interpress master?

Interpress uses the notion of *priority* to resolve the ambiguity. Each object may be assumed to have a numeric priority; when two objects overlap, the object with greatest priority is visible. Much of the time, however, priority is unimportant. If only a single color is used on a page or if no two objects overlap one another, it doesn't matter which objects have high priority. Interpress ignores priority problems unless explicitly told otherwise, since so many images will use only one color—black.

When the relative priority of objects is important, the master must set the imager variable *priorityImportant* (index 5) to a non-zero value. Whenever *priorityImportant* is not zero, the master

may call mask operators to place objects on the page and be assured that objects laid down later will have higher priority than objects laid down earlier. In other words, the order of execution of MASK operators determines the priority.

Using this technique, we can see how Figure 15.8 might have been generated. First, *priorityImportant* is set to 1; then the color is set to gray; then MASKFILL is called to generate the parallelogram; then the color is set to black; then MASKVECTOR is called twice to generate the two black strokes; finally, *priorityImportant* can be set back to 0. Because the black strokes are laid down after the parallelogram, they have higher priority. It's a good idea to leave *priorityImportant* set to 0 whenever possible, since the imager must usually work harder to preserve priority than to ignore it.

Let's consider another example, the box shown in Figure 3.1 that is generated by Example 3.1. We can create this box another way, using priority. We first create a solid rectangular mask using black ink. Then we set the ink to "white" and create a second solid rectangular mask that is slightly smaller than the first one, so as to leave a solid outline 1 mm wide. This technique is illustrated in the following master:

```
--Example 15.8, equivalent to Example 3.1. Uses priority and white ink--
--0--     BEGIN { }
--1--     {
--2--     {
--3--     1 5 ISET              --set priorityImportant--
--4--     0.0249 0.2281 0.1661 0.0264 MASKRECTANGLE
--5--     0 SETGRAY
--6--     0.0259 0.2291 0.1641 0.0244 MASKRECTANGLE
--7--     } DOSAVESIMPLEBODY  --restore color and priorityImportant--
--8--     }
--9--     END
```

This technique is not generally recommended, since it's almost certainly slower to create two large filled rectangles than four strokes, as in Example 3.1. A more interesting and appropriate use of "white" ink prints the lettering in Figure 15.2a.

When priority is used, it's important to be sure that *all* objects whose priority is important are generated when *priorityImportant* has a non-zero value. A change to the page image induced by a mask operator is said to be *ordered* if the mask operator is executed when *priorityImportant* is not zero, and *unordered* if it is zero. Interpress preserves the priority order of all ordered masks, but may alter the priority among unordered masks or between any unordered mask and an ordered mask. To illustrate the use of this rule, consider Example 3.5, which extended Example 3.1 to print the word "Interpress" in the middle of the box. Were we using the technique in Example 15.8 to generate the box, the analog of Example 3.5 would be:

```
--Example 15.9, equivalent to Example 3.5--
-- 0--    BEGIN { [ xerox, xc82-0-0, times ] FINDFONT 0.00635 SCALE MODIFYFONT 0 FSET }
-- 1--    {
-- 2--    {
-- 3--    1 5 ISET              --set priorityImportant--
-- 4--    0.0249 0.2281 0.1661 0.0264 MASKRECTANGLE
-- 5--    0 SETGRAY
-- 6--    0.0259 0.2291 0.1641 0.0244 MASKRECTANGLE
-- 7--    1 SETGRAY
-- 8--    0 SETFONT 0.07366 0.23876 SETXY <Interpress> SHOW
-- 9--    } DOSAVESIMPLEBODY  --restore color and priorityImportant--
--10--    }
--11--    END
```

16

Utility programs

Most users of Interpress will want to use several *utility programs* as well as Interpress printers. Utility programs take as input one or more Interpress masters and create a new master, usually by rearranging or combining the input masters in some way. For example, a utility program could extract certain pages from a master, or combine pages from two masters, and so on. Utility operations are often cascaded several times to produce a finished document. These functions have important uses in environments where several Interpress-creating programs may contribute bits and pieces to a single document.

At first glance, it might appear that Interpress masters are going to be difficult to work on—after all, they may comprise full programming generality, and it's notoriously difficult to write programs that understand and rearrange other programs. However, Interpress has been designed so that many utility functions can be implemented without requiring sophisticated software. Some utilities need only to decipher a master's skeleton in order to identify the preamble and each page body. Others require some form of pseudo-interpretation of the master to detect certain things, such as which frame elements are set in the preamble.

This section explains how a number of utility functions can be carried out on Interpress masters. While the list of functions covered is not exhaustive, the techniques used to achieve them should be suggestive of approaches to other problems as well.

16.1 Notation and assumptions

Since this section makes frequent reference to parts of masters, it's helpful to have a notation that denotes easily the different parts of a master's skeleton. The notation $=M.n=$ will be taken to stand for the sequence of literals in the nth part of master M. $=M.0=$ are the literals in the preamble; $=M.1=$ those in the first page body, and so on. These literal sequences do *not* include the braces surrounding them, { }, to indicate the beginning and end of the body. Thus a complete three-page master A would be written as:

BEGIN $\{=A.0=\}$ $\{=A.1=\}$ $\{=A.2=\}$ $\{=A.3=\}$ END

It is a relatively straightforward operation to scan a master and find the parts of its skeleton. Each token in the encoding must be examined to see if it is "{", "}", "BEGIN", or "END"; the length of each token must also be determined in order to decide how many bytes of data to

skip before looking for the next token. The result of this scanning process is an indication of the position in the file of the beginning and end of each part of the master. This information is used when parts of the master are copied into the output master or rearranged in various ways.

16.2 Selecting pages

Perhaps the simplest utility function is one that extracts from an input master a selected set of pages and creates an output master containing only those pages. The output master simply contains the preamble from the input master, *A*, and each page body that is desired. If pages *i*, *j*, and *k* are extracted, the output master will be:

BEGIN {=*A*.0=} {=*A*.*i*=} {=*A*.*j*=} {=*A*.*k*=} END

This utility is so simple and so fast that Interpress printers provide page selection as a standard printing service (see Section 18). Nevertheless, it's useful to have a utility program that will do this job as well, in order to save communication or storage capacity.

16.3 Selecting pages from two masters

An output master may be created by extracting some pages from master *A* and some pages from master *B* and combining them in the output master. The principle behind this technique is that it is always possible to copy a preamble into each page body that uses it. Thus a master that contains page 1 from *A*, page 1 from *B*, and page 8 from *A* can be formed as follows:

BEGIN { } {=*A*.0= =*A*.1=} {=*B*.0= =*B*.1=} {=*A*.0= =*A*.8=} END

This technique simply inserts the appropriate preamble before each page body. The disadvantage of this technique is that an Interpress printer will have to execute the preamble (*A*.0 or *B*.0) once for each page body, which defeats the purpose of the preamble. An alternative method might be to form:

BEGIN {=*A*.0=} {=*A*.1=} {*ZeroFrame* =*B*.0= =*B*.1=} {=*A*.8=} END

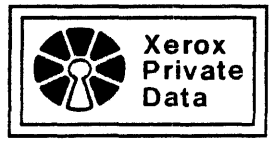
Each page from *B* that is included in the output master will contain both *B*'s preamble and the necessary page body. The notation *ZeroFrame* stands for an Interpress program fragment that is inserted to zero every frame element, since a preamble is allowed to assume that all frame elements are initially zero. It could be defined as:

```
--ZeroFrame macro: zero all elements of the frame--
--0--      0 0 FSET 0 1 FSET 0 2 FSET . . . 0 49 FSET
```

Since *A* and *B* play a symmetric role in this operation, one could equally well generate the master:

BEGIN {=*B*.0=} {*ZeroFrame* =*A*.0= =*A*.1=} {=*B*.1=} {*ZeroFrame* =*A*.0= =*A*.8=} END

The choice of the two forms should probably be based on how many pages of the output come from each input master. If the output contains a single page from *B* and 90 pages from *A*, the first form will be more compact and will be processed more efficiently by printers because most of the pages use the information computed by the =*A*.0= preamble.



16.3.1 Combining two preambles

A master that combines pages from two masters could be described more efficiently if the preambles from both input masters could be combined into a single preamble suitable for the output master. This section shows several ways to go about it.

The basic idea is to evaluate a preamble and then to package into a single vector all its frame elements, the only permanent results a preamble is allowed to have. The packaging is achieved with an Interpress code fragment we shall name *SaveFrame*, which leaves on the stack the packaged vector. The operation *StoreFrame* actually stores this vector in a frame element. Then at the beginning of a page body a *RestoreFrame* operation unpackages the vector into the various frame elements, so that the frame will have exactly the same values in its elements that were computed by the preamble. Several optimizations of these steps are explored after we explain the general case.

First, we define four macro-like sequences that manipulate the frame in various ways:

```
--StartPreamble macro: used at beginning of an original preamble--
-- 0--    {                          --simply start a body--

--EndPreamble macro: package all frame elements into a vector on the stack--
-- 1--    0 FGET 1 FGET 2 FGET . . . 49 FGET
-- 2--    50 MAKEVEC
-- 3--    } MAKESIMPLECO DOSAVEALL --execute the original preamble, restore frame and imager vars--
-- 4--    0 MARK                     --protect vector on stack--

--StoreFrame(n) macro: store packaged frame vector into frame element n--
-- 5--    UNMARK0                    --remove mark--
-- 6--    n FSET

--RestoreFrame(n) macro: unpackage frame vector stored in frame element n--
-- 7--    n FGET
-- 8--    DUP 0 GET 0 FSET
-- 9--    DUP 1 GET 1 FSET
--  --    --. . .--
--10--    DUP 49 GET 49 FSET
--11--    POP
```

Now we can demonstrate how to combine two preambles into one. As an example, consider the problem posed in Section 16.3, to make a master that contains page 1 from *A*, page 1 from *B*, and page 8 from *A*. Example 16.1 shows a master that combines the preambles:

```
--Example 16.1--
-- 0--    BEGIN {
--  --    -- First process A's preamble --
-- 1--    StartPreamble
-- 2--    =A.0=                      --A's preamble--
-- 3--    EndPreamble
--  --    -- Then process B's preamble --
-- 4--    StartPreamble
-- 5--    =B.0=                      --B's preamble--
-- 6--    EndPreamble
--  --    -- Now save packaged frames in frame --
-- 7--    StoreFrame(1)              --B's packaged frame--
-- 8--    StoreFrame(0)              --A's packaged frame--
-- 9--    }                          --end new preamble--
--10--    { RestoreFrame(0) =A.1= }  --a page from A--
--11--    { RestoreFrame(1) =B.1= }  --a page from B--
--12--    { RestoreFrame(0) =A.8= }  --a page from A--
--13--    END
```

Lines 1–3 execute *A*'s preamble, package the resulting frame, and save it on the stack. The *EndPreamble* macro guarantees that any changes made to imager variables by *A*'s preamble will not be seen by *B*'s preamble, because DOSAVEALL restores them. Likewise, it insures that all frame elements are initially zero, since Interpress requires that they be zero when a preamble is executed. Lines 4–6 repeat the same process for *B*'s preamble. Lines 7 and 8 now store these packaged frames into the frame, so that the packages will be available to the individual page bodies. Lines 10–12 show how page bodies are written: at the beginning of each page body, the corresponding frame package is unpackaged and stored in the frame.

An obvious optimization to this procedure is to detect when a preamble does not set all 50 frame elements, and change the *EndPreamble* and *RestoreFrame* operations to package and unpackage only those frame elements actually used by the preamble. To determine which frame elements are used, the utility program must be able to simulate the execution of the preamble.

In particularly simple cases, even greater optimizations are possible. If it should happen to turn out that no frame element is used by both preambles, that the first preamble never sets an imager variable, and that the second preamble never examines the contents of any frame element, the preambles can simply be joined:

BEGIN {=*A*.0= =*B*.0=} {=*A*.1=} {=*B*.1=} {=*A*.8=} END

Or it may turn out that the preamble needed most frequently has a spare frame element in it that can be used to store the packaged form of the other preamble, thus allowing the most frequently used preamble to remain unpackaged. Example 16.2 shows how Example 16.1 could be modified if *A* is to be used most frequently and has frame element 43 available.

```
--Example 16.2--
-- 0--    BEGIN {
--  --    -- First process B's preamble --
-- 1--    StartPreamble
-- 2--    =B.0=                    --B's preamble--
-- 3--    EndPreamble              --package B's frame into a vector on stack--
--  --    -- Then process A's preamble --
-- 4--    =A.0=                    --A's preamble--
--  --    -- Now save packaged frame in frame --
-- 5--    StoreFrame(43)           --B's packaged frame--
-- 6--    }
-- 7--    { =A.1= }                --a page from A--
-- 8--    { RestoreFrame(43) =B.1= } --a page from B--
-- 9--    { =A.8= }                --a page from A--
--10--    END
```

16.4 Merging two pages into one

Suppose that two one-page input masters *A* and *B* already have information positioned properly on both pages, but that the two pages should in fact be printed as one, essentially merging the information from both onto one page. For example, *A* might be the output of a text-processing system, and *B* the output of an illustration-making system; assume that the text system leaves room for the illustration and that the illustration system positions the illustration properly. If the preambles are not combined, we can achieve the merged output as follows:

```
--Example 16.3--
--0--     BEGIN
--1--     { =A.0= }                  --A's preamble is output preamble--
--2--     {                          --page body--
--3--     { ZeroFrame =B.0= =B.1= } MAKESIMPLECO DOSAVEALL  --print B's page--
--4--     =A.1=                      --and then print A's page--
--5--     }                          --end page body--
--6--     END
```

This example shows how DOSAVEALL is used to insure that the state of imager variables at the beginning of line 4 will be the same as at the beginning of line 3. For example, this makes sure that if $=B.0=$ and/or $=B.1=$ change the current transformation, it will be restored to its original value before $=A.1=$ is executed. There are other possible arrangements that will have the same effect, such as:

```
--Example 16.4--
--0--     BEGIN
--1--     { =A.0= }                  --A's preamble is output preamble--
--2--     {                          --page body--
--3--     { =A.1= } MAKESIMPLECO DOSAVEALL  --print A's page--
--4--     ZeroFrame =B.0= =B.1=      --and then print B's page--
--5--     }                          --end page body--
--6--     END
```

As before, the roles of *A* and *B* can be interchanged in either of these examples.

The techniques described in Section 16.3.1 for merging preambles can also be used in programs that merge pages.

16.4.1 Priority

If either of the masters sets *priorityImportant*, the process of merging information may be more complicated. If the images created by the two masters do not overlap on the page, then there will be no problem. If there is overlap, however, someone must determine which master is to have priority. Then the merged master can be constructed by turning on *priorityImportant* and placing first the original page body that should have low priority:

```
--Example 16.5--
--0--     BEGIN
--1--     { =A.0= }                  --A's preamble is output preamble--
--2--     {                          --page body--
--3--     1 6 ISET                   --set priorityImportant--
--4--     { =A.1= } MAKESIMPLECO DOSAVEALL  --print A's page, low priority--
--5--     ZeroFrame =B.0= =B.1=      --and then print B's page, high priority--
--6--     }                          --end page body--
--7--     END
```

This problem will probably not arise frequently, since priority is infrequently used, and since it will be rare that masters to be merged will overlap on the page in such a way that it's important to establish relative priority.

16.5 Applying a geometric transformation

A common utility function is to change the geometry of one or more pages by applying a transformation. We can insert a program fragment at the beginning of a page body to modify the current transformation to change the geometry of the page.

For example, suppose we have a master that is formatted for printing on one side of each sheet of paper but we want to print it on both sides of the paper and to reposition each page away from the binding edge so that holes can be punched in the paper. So images on odd pages, beginning with page 1, must be moved slightly to the right, and those on even pages slightly to the left. Let's assume these displacements are to be 1 cm. The following master will achieve this result:

```
--Example 16.6--
--0--     BEGIN
--1--     { =A.0= }                                    --keep preamble--
--2--     {  0.01 0 TRANSLATE CONCATT =A.1= }          --translate odd page right--
--3--     { -0.01 0 TRANSLATE CONCATT =A.2= }          --translate even page left--
--4--     {  0.01 0 TRANSLATE CONCATT =A.3= }          --translate odd page right--
--5--     { -0.01 0 TRANSLATE CONCATT =A.4= }          --translate even page left--
-- --     . . .                                        --more pages--
--6--     END
```

This example illustrates the power of the current transformation. Since all of the geometry in each page body is transformed by the current transformation, the geometry of the entire page can be modified simply by altering the current transformation appropriately.

Note that in general the creator of a master must know whether it will be printed on both sides of the paper and will format it accordingly. For example, page numbers would normally alternate from right to left corners and line measures would be reduced to provide the same visible margins in spite of the binding.

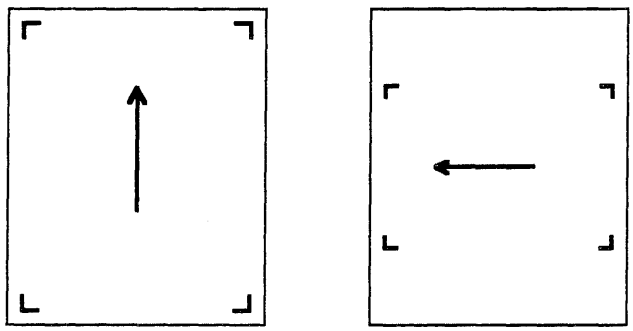


Figure 16.1. Original page and desired result.

The same technique can be used to scale and/or rotate the image. For example, suppose we start with a page that contains an image 7½×10 inches centered in an 8½×11 inch page and we wish to "turn" it so that it can be viewed from the right side, so that there will still be ½ inch margins, and so that the original shape factor is retained (Figure 16.1).

We observe that for ½ inch margins, the "height" of the new image will be 7½ inches, so the original image must be scaled by 7½/10=0.75 to obtain the new image. The coordinate transformations are worked out by observing Figure 16.2. We have:

T_{AB} = ⟨−0.0127 −0.0127 TRANSLATE⟩
T_{BC} = ⟨0.75 SCALE⟩
T_{CD} = ⟨90 ROTATE⟩
T_{DE} = ⟨0.2032 0.0682625 TRANSLATE⟩
T_{AE} = ⟨T_{AB} T_{BC} T_{CD} T_{DE} CONCAT CONCAT CONCAT⟩

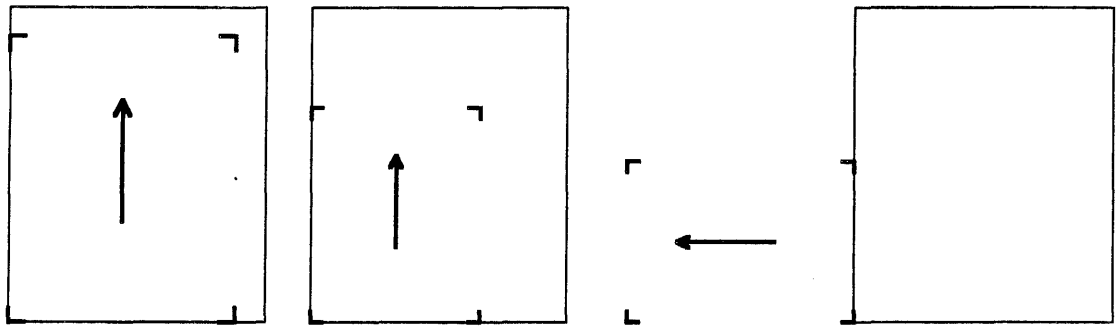


Figure 16.2. Intermediate transformations.

The master becomes:

```
--Example 16.7--
-- 0--    BEGIN
-- 1--    { =A.0= }              --keep preamble--
-- 2--    {                      --start page body--
-- 3--    -0.0127 -0.0127 TRANSLATE
-- 4--    0.75 SCALE
-- 5--    90 ROTATE
-- 6--    0.2032 0.0682625 TRANSLATE
-- 7--    CONCAT CONCAT CONCAT CONCATT
-- 8--    =A.1= }                --A's original page body--
-- 9--    }                      --end of page body--
--10--    END
```

This is not the only way to achieve this transformation. It turns out that it can be done with only three primitive transformations. What are they?

16.5.1 Suitable transformations

While it is easy to apply arbitrary geometric transformations to entire pages, some caution is required. Since a printer may restrict the sizes and rotations of text characters it can show or of pixel arrays it can image, arbitrary transformations may yield unprintable masters. The restrictions a printer imposes are described by the *easy net transformations* for a font (see Section 13.5 and § 4.9.3) and the easy net transformations for pixel arrays. A common convention for printers configured to print computer output is to allow 60% scale factors when text is rotated 90 degrees so that "two up" pages can be printed (see Section 16.7). Of course, arbitrary translation transformations are always acceptable to a printer.

16.6 Merging and positioning

The transformation techniques illustrated in the previous section (16.5) can be used in conjunction with other utilities. For example, a text document can have an illustration merged in, after appropriate sizing and positioning. This utility uses a combination of the transformation techniques shown in Section 16.5 and the merging schemes of Section 16.4.

Suppose that a text document is represented by master *A* and a one-page master *B* contains an illustration for page 2 of *A*. The transformation T_{pos} represents the sizing and positioning required of *B* to fit appropriately on *A*'s page 2. The overall structure of the merged master is similar to that of Example 16.3. We have:

```
--Example 16.8.--
-- 0--    BEGIN
-- 1--    { =A.0= }              --keep A's preamble--
-- 2--    { =A.1= }              --A's page 1--
-- 3--    {                      --start page body for page 2--
-- 4--     { ZeroFrame =B.0=     --B's preamble--
-- 5--     Tpos                  --put here code to create transformation--
-- 6--     CONCATT
-- 7--     =B.1=                 --B's original page body--
-- 8--     } MAKESIMPLECO DOSAVEALL
-- 9--    =A.2=                  --print A's page--
--10--    }                      --end of page body--
--11--    { =A.3= }              --rest of A's pages--
--12--    END
```

A particularly interesting case of merging an illustration arises in teacher's editions of textbooks. Sometimes each page of the teacher's edition contains, in an upper corner, a small image of a page from the textbook, a couple of inches on a side. This effect can be achieved easily with Interpress: to create the teacher's edition master, appropriately scaled and translated copies of page bodies from the textbook master are merged into each page of the teacher's edition.

The comments in Section 16.3.1 about merging preambles and in Section 16.4.1 about priority also apply to the cases discussed in this section.

16.7 Imposition

Transformation and merging techniques may be combined to perform *imposition*, a term used in the printing industry to denote the arrangement of several page images on a single piece of paper so that the folding and trimming of the paper results in a book with the proper page order. Often 16 or 32 page images are imposed on a single large sheet. With small paper sizes, two or four page images are all that will fit.

By way of illustration, let us consider "two-up signature" format, in which two pages' worth of text are printed on each side of each piece of paper, shrunk down to about half their former size and rotated 90 degrees (see Figure 1.7). A utility to create such a master uses both transformation and merging techniques.

The example below shows how a master *A* might be rearranged to appear in two-up form. The transformations T_{top} and T_{bottom} are the transformations used to position the top image and the bottom image respectively on the output page.

```
--Example 16.9.--
-- 0--    BEGIN
-- 1--    { =A.0= }                --keep A's preamble--
-- 2--    {                        --begin page body--
-- 3--      { Tbottom CONCATT =A.1= } MAKESIMPLECO DOSAVEALL
-- 4--        Ttop    CONCATT =A.2=
-- 5--    }                        --end page body--
-- 6--    {                        --begin page body--
-- 7--      { Tbottom CONCATT =A.3= } MAKESIMPLECO DOSAVEALL
-- 8--        Ttop    CONCATT =A.4=
-- 9--    }                        --end page body--
--  --    . . .                    --more page bodies as required--
--10--    END
```

We can easily derive examples of T_{top} and T_{bottom}. If the original 8½×11 inch page is to be scaled without changing the relative sizes of margins, the scale factor will be 5½/8½, because

the horizontal dimension of the original page will have to fit it 11/2=5½ inches. Both images must be rotated 90 degrees. The origin for the "top" image will lie at x=11×(5½/8½) inches = 0.18079 meter, y=5½ inches = 0.1397 meter. The origin for the "bottom" image will lie at the same x, but y=0. So we have:

T_{top} = ‹11/17 SCALE 90 ROTATE 0.18079 0.1397 TRANSLATE CONCAT CONCAT›
T_{bottom} = ‹11/17 SCALE 90 ROTATE 0.18079 0 TRANSLATE CONCAT CONCAT›

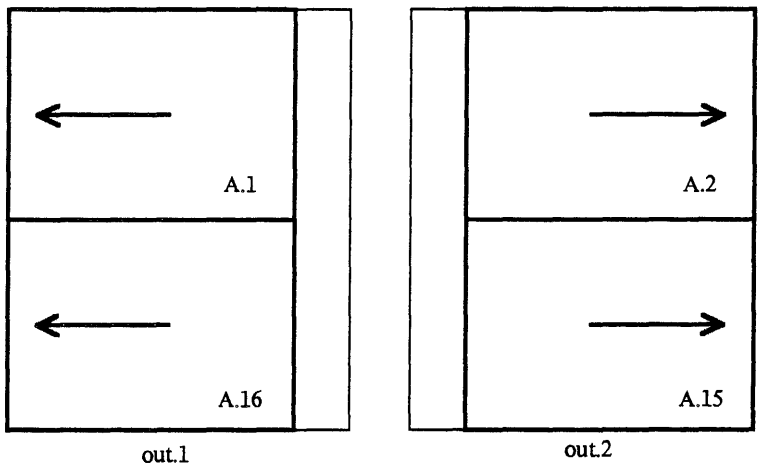


Figure 16.3. Two pages from a signature.

There are many variations on this theme. The transformations might be chosen differently so that extra "gutter space" is left between the two pages. The transformations on even and odd resulting pages might be different, as in Example 16.6, so that when the document is printed on both sides of the page the front and backside images would align. The page order might also be chosen so that the final pages could be saddle-stapled to yield a document that reads properly. For example, if *A* were a 16-page document, the first output page would have *A*.1 on top and *A*.16 on the bottom; the second output page, to be printed on the back of the first, would have page *A*.2 on top and page *A*.15 on the bottom, and so forth. Moreover, for this kind of binding to work, odd pages will have to use transformations that rotate pages in the opposite direction (see Figure 16.3).

16.8 Embedding information in masters

One problem faced by document-assembly utilities is obtaining the information necessary to control the merging of masters. This problem is best illustrated with a simple example: merging a single illustration into a document such as this one. The problem is, how does the utility program know what to merge onto which page and where it should lie? Clearly, this information must be derived and communicated to the utility program.

Placement information can be derived in several ways. The easiest way is to assume that one of the masters, say the one that contains the document's text, will be the controlling master. The composition system that creates the master figures out how big the illustration is to be—more on this below—and allocates sufficient space on an appropriate page. As a result of this layout

decision, the composition system must tell the merge utility to "place 'figure12-2' on page 14 at (0.103, 0.0467)," where 'figure12-2' is the name of a one-page master that contains the illustration and the coordinates identify the location of the bottom center of the illustration, in meters. If the illustration were to be scaled or rotated, other information must be provided as well.

There are several possibilities for communicating these merging instructions to the utility program. They could be placed in a separate file, which the utility reads along with the master. Alternatively, they can be embedded within the master itself as comments in the encoding. That is, the instructions may be placed within *sequenceComment* tokens (§ 2.5.2). These comments will be ignored by an Interpress printer, but can be parsed and interpreted by a merge utility. Of course, it will be necessary to specify the syntax and semantics of merge instructions and to guard against comments included for other reasons interfering with merge instructions.

Another possibility is to place instructions at the end of an Interpress master, after the final END token. Like instructions embedded within comments, an Interpress printer will ignore extra information at the end of a master.

A few simple conventions will make it easy to locate the instructions at the end of the master. The last four bytes of the file, for example, could be a "password" or unique code that indicates that merging instructions have been appended to the file. None of these four bytes should have the value 103 decimal, since that value will always be the last byte of an Interpress master (§§ 2.5.1 and 2.5.4). Immediately preceding these four bytes might be four bytes that are interpreted as an integer that gives the number of bytes of merging instructions. This information then allows the beginning of the merging instructions to be located. Moreover, many blocks of data of this sort, each followed by a length and a password, can be appended to the master. By working backwards, finding first the password and then the length of each block, a program can locate all of the appended information.

Embedded information can be used for many different purposes. For example, an illustration system might embed information about the size of an illustration—the left, right, bottom, and top coordinates of a box that completely surrounds the illustration—in the master it generates. This information could be used by a layout system or text formatter to leave enough space for the illustration and by the merge utility to prepare the coordinate transformation that will place the illustration according to the merge instructions.

16.9 Routing sheets

If several copies of a document are to be printed and routed to separate individuals, it may be convenient to preface each copy with a cover sheet that identifies the recipient. If copies are to be mailed, the cover sheet might be formatted so that it could be inserted into a window envelope or so that the entire document could be folded, stapled, and mailed.

The technique for obtaining a cover sheet customized for each copy was illustrated in Section 12.2.1, using the IFCOPY operator. The utility program we seek simply inserts before the first page of the original master a new page that uses the IFCOPY operator to print customized routing sheets. The following example illustrates how this might be done:

```
--Example 16.10.--
-- 0--    BEGIN
-- 1--    { =A.0= }                  --keep A's preamble--
-- 2--    {                          --page body for routing sheet--
-- 3--    0.000035278 SCALE CONCATT     --units of 1/10 point--
-- 4--    [ xerox, xc82-0-0, times ] FINDFONT 120 SCALE MODIFYFONT 0 FSET --get font--
-- 5--    0 SETFONT
--  --                               --define composed operator for printing addresses--
-- 6--    {                          --PrintAddress(name, mailstop)--
-- 7--    2880 5040 SETXY
-- 8--    SHOW                       --print mail stop--
-- 9--    2880 5220 SETXY
--10--    SHOW                       --print name--
--11--    } MAKESIMPLECO 1 FSET      --save composed operator in frame[1]--
--  --                               --now for calls for individual recipients--
--12--    { POP 1 EQ } MAKESIMPLECO
--13--    { <John G. James> <A/21-13> 1 FGET DO } IFCOPY
--  --
--14--    { POP 2 EQ } MAKESIMPLECO
--15--    { <Robin Carruthers> <Admin. 14> 1 FGET DO } IFCOPY
--  --                               --as many as you want--
--16--    }                          --end of page body for routing sheet--
--17--    { =A.1= }                  --now copy in A's page bodies--
--18--    { =A.2= }
--19--    { =A.3= }                  --as many page bodies as A has--
--20--    END
```

16.10 Closure

The device-independence of a master can be increased by a utility program that "closes" it. Closing means reducing or eliminating the master's references to the environment. If a master contains no references to the environment it will be insensitive to the configuration of font libraries and forms on the printer.

The most important function a closure utility must perform is to copy into the output master the contents of any *sequenceInsertFile* requests (Section 11.2). The utility will need to know the syntax of file names used for *sequenceInsertFile* and will need to have access to the same files that the printer can obtain.

A closure utility can also copy into the master the definitions of fonts that it uses, building composed operators such as those illustrated in Section 14.4. To make the master, the utility will need to have access to a library of font definitions. This step will result in a master that makes no references to the environment, but will be quite bulky. Moreover, the device-independence of the result may not increase, since some printers may be incapable of high fidelity renderings of the graphics required for letterforms.

Document closure is most useful for archival storage. When a document is printed from an archive, the environment and font library may have changed since the document was created. By closing a document before archiving it we ensure that it will be printed in the environment for which it was created.

17

Hints for the creator

This section contains a number of suggestions for the design of programs that create Interpress masters. These suggestions and hints do not introduce new aspects of Interpress, but rather summarize observations made earlier.

17.1 Do's and dont's

Don't view Interpress as a general-purpose programming language. The programming-like aspects of Interpress are provided so that arguments can be passed to operators and so that composed operators can be formed, not so that long, complex computations can be undertaken. Generally, the creator is better suited to computing than is the printer. Although some printers will have ample computing power, most will be optimized for generating images, not for general computation. Moreover, printers are likely to charge for computing time as well as for paper and other consumables required to print a master.

Whenever possible, use the simplest Interpress form available to specify a page. This practice will allow the master to be printed on the widest variety of printers and may cause it to be printed more efficiently. Consult the discussion of *subsets* (§ 5.1.1 or Section 19) to determine those features of Interpress that are most widely supported.

Set up a page coordinate system with a fine enough scale so that masters can use integers rather than rationals in coordinates. (Section 6.2)

Use the preamble to extract values from the environment and save them in the initial frame. References to the environment are likely to be much more expensive than references to the frame, so they should be made infrequently. Also, the printer can optimize the job of obtaining fonts from the environment if fonts are specified only in the preamble.

Use CORRECT freely to protect against changes in line length brought on by font approximation and tuning, but try to have access to the right character widths. Correction is relatively expensive when the tolerance is exceeded, and moreover the corrective steps taken may spoil the appearance of the image.

Don't use CORRECT when line length changes due to font approximation can be tolerated, as in many computer-printing applications.

Don't use CORRECT as the principal means to achieve line justification, but only to compensate for font approximations in an already-justified line.

Use relative positioning (SETXYREL, SETXREL, SETYREL) when the relationship between two adjacent objects is important. Characters within a word and words within a line should be positioned relative to each other.

Use absolute positioning (SETXY) to locate unrelated or loosely-related objects on the page. Thus the overall position on the page of an entire line of text should be set with global positioning, though the spacing within that line should be relative.

Don't use more that 250 relative positioning commands (SETXYREL, SETXREL, SETYREL) between absolute positionings (SETXY). Since each character operator calls a relative positioning operator, this means that fewer than 250 characters should be SHOWn between absolute positioning commands.

Be sure to use sufficient precision in calculating the measure of text lines (Section 10.4.2).

Don't let matrix concatentation nest too deeply, lest computation errors accumulate. A depth of 8 is completely safe even for high precision.

Don't create an excessive number of composed operators. While they save space in the master, they require space and computation time at the printer. A master is not intended to be a structured program.

If a value is needed in only one place, supply it as a literal in that place, rather than storing it as an element in the frame. Even if it is used in several places, it is better to supply it each time if it is a number; only larger values like vectors and transformations are worth saving.

Take advantage of the saving and restoring facilities of DOSAVE, DOSAVEALL, and DOSAVESIMPLEBODY to save and restore imager variables. This is especially useful when changes may be nested inside other changes, e.g., for the current transformation, *priorityImportant*, and *correctPass*. The correct procedure is to save the current setting, change the variable appropriately, execute the code that requires the change, and restore the original setting. This allows changes to nest nicely.

Don't set imager variables measured in device coordinates directly with ISET except when saving and restoring them. Use the functions SETXY, SETXYREL, SETXREL, SETYREL, SETCORRECTTOLERANCE, and SETCORRECTMEASURE instead.

Don't modify the current transformation except with CONCATT and saving/restoring operations.

Don't set *priorityImportant* to a non-zero value unless necessary.

Don't use non-uniform scaling (SCALE2) unless you know what you're doing. While it may be tempting in some applications to set up a page coordinate system that uses different units in x and y, beware that all calls on character operators will propagate the effects of the different units so that characters will appear elongated or heightened. This distortion can be compensated by an appropriate non-uniform scaling transformation passed to MODIFYFONT when the font is defined. Distortions to graphical strokes, rectangles, and pixel arrays, however, cannot be compensated easily.

17.2 Good Interpress style

In order to present material in an easily-understood order, not all examples in this Introduction use "good Interpress style." This section lists, for each example, suggestions for improving its style. Stylistic suggestions fall into five categories:

P A page coordinate system should be used, so that coordinates in the page bodies can be represented using integers (see Sections 5.3.4 and 6.3).

F Fonts should be set up in the preamble and saved in frame elements. Then SETFONT is used in a page body to pass the font to the imager routines.

X SETXREL can be used rather than SETXYREL when $y=0$.

S Saving and restoring state can be achieved with DOSAVESIMPLEBODY. This suggestion is accompanied by references to appropriate line numbers in the example, in parentheses.

Table 17.1 Stylistic flaws in examples

Example	P	F	X	S	Comments
3.1	•				
3.2	•	•			
3.4	•	•			
3.5	•	•			
3.6	•	•			
3.7	•	•			
4.1	•				
6.2	•				
6.6				(8, 12)	
9.1			•		
9.2			•		
9.3			•		
10.1					Absolute character positioning should be avoided.
10.7				(1, 3)	
10.8				(1, 4)	
10.10				(1, 3)	
10.12				(2, 6)	
10.13				(2, 4, 8, 9)	
10.14				(2, 4, 8, 12)	
10.15				(14, 17)	
14.9			•		
14.10			•		
15.1	•				
15.8	•				
15.9	•				

17.3 Miscellaneous techniques

This section presents a number of techniques that may be useful in preparing masters, but may not be obvious from the preceding discussion of Interpress.

17.3.1 Adaptive transformations

It is possible to compose a master without knowing the physical size of the medium on which it will be printed. The idea is that at the beginning of each page body, some Interpress code will interrogate imager variables to determine the size of the medium and set the current transformation accordingly.

Let's illustrate this technique. Suppose that a page body is prepared assuming the origin (0, 0) will be in the lower left corner and the point (*xmax*, *ymax*) will be in the upper right corner. The following mixture of Interpress and Pascal-like code alters the current transformation so that the page's coordinate system will map into the Interpress coordinate system:

```
fieldYSize: = fieldYMax−fieldYMin;
fieldXSize: = fieldXMax−fieldXMin;
mediumAspect: = fieldYSize/fieldXSize;
pageAspect: = ymax/xmax;
if mediumAspect > pageAspect then scale: = xmax/fieldXSize
    else scale: = ymax/fieldYSize;
<   −xmax/2 −ymax/2 TRANSLATE
    scale SCALE
    fieldXMin+fieldXSize/2 fieldYMin+fieldYSize/2 TRANSLATE
    CONCAT CONCAT CONCATT
>;
```

This code interrogates imager variables to find out the dimensions of the *field*, the region of the medium on which an image can be placed (§ 4.3.1). By comparing the aspect ratios of the medium and of the page to be printed, it determines whether the width or the height will be the controlling dimension that determines the overall scale factor. Then a transformation is constructed to center the page within the field. The transformation is formed in three steps: the page body will be translated so that the origin becomes the center of the page, then the entire page will be scaled, and finally translated so that the origin is placed at the center of the field. This procedure could be translated into an Interpress composed operator and executed at the beginning of each page body.

A variant of this procedure could rotate the page if necessary to obtain a favorable aspect ratio.

17.3.2 Obtaining a character's width in the master

Although the character metrics are not directly accessible to the master, it is possible to determine a character's width. Since a side effect of SHOWing a character is to change the current position, we can determine a character's width by examining the change. Moreover, we can prevent the character from actually being printed by using the imager variable *noImage* (§ 4.8). The following example illustrates the idea:

```
--Example 17.1. Assume SETFONT has set proper font.--
--0--     2160 3400 SETXY      --set current position to someplace reasonable--
--1--     {
--2--     1 14 ISET            --set noImage=1--
--3--     MOVE                 --change current transformation so origin at current position--
--4--     <c> SHOW             --show the character--
--5--     GETCP                --compute current position, in current coordinate system--
--6--     } DOSAVESIMPLEBODY   --protect against permanent imager variable changes--
--7--                          --stack has <widthX, widthY> in current coordinate system--
```

This example will measure the character width in whatever units are used for the current coordinate system.

17.3.3 Overriding character widths

Sometimes a master wishes to use the graphic images from a font but to override the widths associated with each character. The following example shows how a single character might be imaged:

```
--Example 17.2. Assume SETFONT has set proper font.--
--0--     {
--1--     <c> SHOW                  --show the character--
--2--     } MAKESIMPLECO DOSAVEALL --restore current position, CORRECT state
--3--     xWidth yWidth SETXYREL    --put your widths here--
--4--     CORRECTMASK               --call to CORRECT machinery--
```

Variants of this example would be required for space characters, since they must call CORRECTSPACE (Section 9.3.2). Note that the *x* and *y* arguments to SETXYREL are in the coordinate system current when the code above is executed, not in the character coordinate system.

If character widths are to be routinely overridden, it is possible to define a new font by giving a definition of each character operator as a composed operator that prints a character from another font (the "real" one) and then sets the width appropriately. A character operator in this font might look like:

```
--Example 17.3.--
--0--     {
--1--     {
--2--      1 SETFONT                --set the font to the "real" font--
--3--      [ 13 ] SHOW              --show the character corresponding to this operator--
--4--     } MAKESIMPLECO DOSAVEALL --restore font, current position, CORRECT state
--5--     xWidth yWidth SETXYREL    --put your widths here--
--6--     CORRECTMASK               --call to CORRECT machinery--
--7--     } MAKESIMPLECO
```

If font 1 is defined with a transformation of <1 SCALE>, then the widths on line 5 will be given in the character coordinate system. An appropriate scaling transformation can be applied to the font of which the composed operator in Example 17.3 is a part.

17.3.4 Obtaining more frame space

For particularly complicated masters there will not be enough space in the frame to hold all the fonts required. In this case, the fonts can be packaged into a Vector, which is then stored in a single frame element. The GET operator can then be used to obtain individual elements of the vector. This idea is exemplified below:

```
--Example 17.4.--
-- 0--    BEGIN {                          --begin preamble--
-- 1--    . . .                            --fill up frame elements 0-49 with fonts 0-49--
-- 2--    0 FGET 1 FGET 2 FGET . . . 49 FGET     --put all 50 fonts on the stack--
-- 3--    50 MAKEVEC 0 FSET                --make a 50-element vector and save in frame[0]--
-- 4--    . . .                            --fill up frame elements 1-49 with fonts 50-99--
-- 5--    }
-- 6--    {                                --begin page body--
-- 7--    14 SETFONT                       --set current font to font 63--
--  --    . . .
-- 8--    0 FGET 38 GET 12 ISET            --set current font to font 38--
--  --    . . .
-- 9--    }
--10--    END
```

17.3.5 Using local variables in the stack

Although the stack is normally used only to pass arguments to operators and not to store local variables, the stack operators can be used so as to give frame-like access to the stack. In particular, we can define "stack-get" and "stack-set" operations, similar to FGET and FSET. These operations are defined below not as operators or as composed operators, but rather as sequences of Interpress program that can be inserted in a master to obtain the desired effect. *StackGet(j)* retrieves the element *j* deep in the stack and places a copy on the top of the stack. *StackSet(v, j)* sets the element *j* deep in the stack to *v*.

$\langle x_j$: Any$\rangle$...$\langle x_1$: Any$\rangle$ *StackGet* → $\langle x_j$: Any$\rangle$...$\langle x_1$: Any$\rangle$ $\langle x_j$: Any$\rangle$

where the element *j* deep in the stack is copied onto the top of the stack. This effect can be achieved with the Interpress program $\langle j$ 1 ROLL DUP $j+1$ *j* ROLL$\rangle$.

$\langle x_j$: Any$\rangle$...$\langle x_1$: Any$\rangle$ $\langle v$: Any$\rangle$ *StackSet* → $\langle v$: Any$\rangle$ $\langle x_{j-1}$: Any$\rangle$...$\langle x_1$: Any$\rangle$

where the element *j* deep in the stack (not counting *v*) is replaced by the value *v*. This effect can be achieved with the Interpress program $\langle j+1$ *j* ROLL *j* 1 ROLL POP$\rangle$.

17.3.6 Loops

Although the base language does not contain iteration constructs explicitly, recursive functions may be used to obtain the effect of loops. For example, the definition of *EQN in § 2.4.8 can be restated as:

```
--Example 17.5.--
-- 0--    --return 1 if <a b EQ> --
-- 1--    2 COPY EQ { POP POP 1 } IFELSE {
-- 2--    --return 0 if not both vectors--
-- 3--    2 COPY TYPE 3 EQ EXCH TYPE 3 EQ NOT { POP POP 0 } IFELSE { 0
-- 4--    -- define compare(a, b, i, op) = 1 if a and b have same length and --
-- 5--    --      a[i+a.l]=b[i+b.l] for i=i..a.n-1 --
-- 6--    { 0 FSET 1 FSET 2 FSET 3 FSET       -- frame[0]=op, frame[1]=i, frame[2]=b, frame[3]=a --
-- 7--      3 FGET SHAPE 4 FSET 5 FSET        -- frame[4]=a.n, frame[5]=a.l --
-- 8--      2 FGET SHAPE 6 FSET 7 FSET        -- frame[6]=a.n, frame[7]=b.l --
-- 9--      4 FGET 6 FGET EQ { 0 } IFELSE {   -- return 0 if a.n=b.n --
--10--      1 FGET 5 FGET GE { 1 } IFELSE {   -- return 1 if i>a.n, i.e., compares are done --
--11--      3 FGET 1 FGET 5 FGET ADD GET 3 FGET 1 FGET 7 FGET ADD GET
--12--        EQ NOT { 0 } IFELSE {           -- return 0 if a[i+a.l]=b[i+b.l] --
--13--      3 FGET 2 FGET 1 FGET 1 ADD 0 FGET 0 FGET DO -- return compare(a,b,i+1,compare) --
--14--      } IF } IF } IF
--15--    } MAKESIMPLECO DUP DO
--16--    } IF } IF
```

This definition suggests how we could make a general *iterator* composed operator. We shall define a routine *iterate(a, b, op, iterate)* which executes $\langle i$ *op* DO$\rangle$ for $a \leq i \leq b$. Because *iterate*

must call itself, it must be passed itself as an argument (there's no way to place the composed operator *iterate* in its own initial frame).

```
--Example 17.6.--
-- --    -- define iterate(a, b, op, iterate) = --
-- --    --   if a>b then return; op(a); iterate(a+1, b, op, iterate) --
-- 0--   { 4 2 ROLL 2 COPY   GT              --stack is op,iterate,a,b,(a>b) --
-- 1--     { POP POP POP POP } IFELSE
-- 2--     { EXCH 4 1 ROLL                   --stack is iterate,b,a,op --
-- 3--       2 COPY DO                       --call op(a)--
-- 4--       4 3 ROLL 1 ADD EXCH             --stack is op,iterate,a+1,b --
-- 5--       4 2 ROLL                        --stack is a+1,b,op,iterate --
-- 6--       DUP DO } IF                     --call iterate(a+1,b,op,iterate)--
-- 7--   } MAKESIMPLECO
```

17.3.7 Rounding to a device coordinate

Although *DROUND (§ 4.3.5) cannot be called directly from the master, the same effect can be obtained with the following code:

```
--Example 17.7.--
-- --     -- x y *DROUND => X Y --
--0--     { 1 ISET 0 ISET                   --save x,y in current position--
--1--       TRANS                           --translate origin to rounded current position--
--2--       0 0 SETXY                       --set current position to that origin--
--3--       0 IGET 1 IGET                   --place X Y on stack--
--4--     } MAKESIMPLECO DOSAVEALL          --protect against all imager variable changes--
```

18

Printing instructions

When an Interpress master is presented to a printer, it is usually accompanied by some *printing instructions* that tell the printer exactly what to do with the master. Examples of printing instructions are the number of copies to print, what kind of paper to use, and what account to charge. In most cases, these instructions are as vital to the printing of a document as the Interpress master itself.

Printing instructions appear in two places. A collection of *master instructions* is encoded as part of the master itself. Additional *external instructions* are sent to the printer as part of a printing request. The Interpress printer merges the two sets of instructions, fills in certain printer defaults if necessary, and then obeys the instructions.

Interpress does not define how external instructions are communicated to a printer. Xerox printers attached to the Ethernet will use a *Printing Protocol* to request printing services, to transmit printing instructions, and to interrogate the progress of a printing request [28].

Some printing instructions are used to fill in a *break page* that the printer may provide as a cover sheet for the document being printed and as a way of separating the output from successive printing jobs. The break page often shows the document name, creation date, printing date, identity of the printer, name of the person who is to receive the document, optional comments, messages describing errors encountered during printing, and so on. The layout of the break page is controlled by the printer. Many devices have no way of physically separating output from different masters, and it is therefore important to have an easily recognized break page.

This section gives examples of the most common printing instructions that apply in an office or computing environment. More complex needs require understanding the details of printing instructions, covered fully in § 3.3.

18.1 Standard instructions

Interpress defines a set of standard printing instructions. Some of the instructions are associated with the master itself, and not with the printing request. These are generally prepared when the master is created, and include such things as the document name, the creation date, the size and type of paper to use to print it, and so on. Others, such as the number of copies to print, are usually part of the printing request.

The list below is divided into *master* and *external* instructions, although Interpress does not dictate which instructions may be provided with the master or with the request. The details of the interpretation of these instructions are given in § 3.3.3.

18.1.1 Master instructions

- Document name (*docName*). A text string identifying the document is a very helpful part of the break page. In computer printing applications, this name might be the filename under which the document is stored in the computer's file system.

  Note that a printing instruction must identify a font (or at least a character set) that can be used to print the document name (*breakPageFont*); otherwise document names in EBCDIC and ASCII will be confused. This same comment applies to other "text strings" in printing instructions.

- Creator (*docCreator*). A text string that gives the name of the program or person who created the master.

- Creation date (*docCreationDate*). A text string giving the date and time when the document was created.

- Message (*docComment*). A text string to be printed on the break page.

- Media sizes and types (*media*). This instruction indicates the size and type of media for which the document has been formatted. Interpress allows different pages to assume the presence of media of different sizes or types, but a printing instruction must instruct the printer which medium should be used for which pages.

- Selection of media by page (*mediaSelect*). This instruction associates with each page a medium to carry its image.

- Selective omission of pages when printing one-side only (*onSimplex*). The creator may indicate blank pages that can be omitted when printing on one side only, but which must be included if printing on both sides.

- Finishing (*finishing*). The master may have been constructed to be used in conjunction with certain kinds of finishing, such as binding, hole-drilling, stapling, etc. This instruction is often part of a printing request rather than a master.

- Duplex (*plex*). This instruction will tell the printer that the document has been formatted for printing on both sides of the paper.

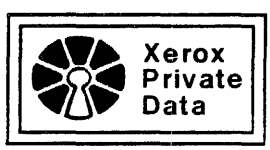
- Hold (*docPassword*). If the document has special protection associated with it, an instruction may indicate a password that must be supplied at the printer site before the document will be printed. The idea is that a person with appropriate privileges must be present at the printer before printing will begin.

18.1.2 External instructions

- Printed for (*jobRecipient*). A text string that identifies the person who should receive the printed document. The font assumed for the string is the same as for the document name.

- Printed by (*jobSender*). A text string that identifies the person who initiated the printing request.

- Account (*jobAccount*). The name and perhaps password of an account that should be charged for printing the document.



- Copies (*copySelect*). The number of copies to print. To work in conjunction with the IFCOPY operator, ranges of copy number are specified, e.g., 10-13, 15.
- Copy name (*copyName*). The creator may associate identifiers with different copies to be printed. This name is available to the IFCOPY operator (Section 12.2.1 and § 2.4.7).
- Priority (*jobPriority*). This instruction controls the priority that the printing job will receive.

Some of the external instructions overlap with master instructions:

- Message (*docComment*).
- Media sizes and types (*media*). While the master might specify the media sizes, the request might specify the media types, e.g., blue paper.
- Finishing (*finishing*).
- Duplex (*plex*).
- Hold (*jobPassword*).

Some of the instructions may specify simple "utility" functions that a printer is willing to handle. For example:

- Selected pages (*pageSelect*). This instruction gives a set of page ranges to be selected from the master and printed.
- Image shift (*xImageShift*). When a document is being printed on both sides of the paper, it's often convenient to shift the image on odd pages to the right and on even pages to the left so as to leave more room for drilled holes or binding. This printing instruction tells the printer to apply such a shift and gives the desired shift amount.

18.2 Encoding instructions

Master instructions are inserted into the master in a body called the *instructions body*, which precedes the master's BEGIN token. This body is executed to obtain one or more property vectors that encode the master instructions; these are then merged with the external instructions. Thus a master with instructions looks like:

{ instructions body } BEGIN {preamble} {page body} {page body} END

The creator may wish to divide the master instructions into two classes:

1. *High priority* instructions that should override any external instructions, e.g., the document name, the creation date, the document password.
2. *Low priority* instructions that may be overridden by the external instructions, e.g., the size and type of paper to use.

The conventional way to encode these instructions is to use the following template:

```
--0--   {                    --beginning of instructions body--
--1--   --construct a property vector for low priority instructions--
--2--   EXCH
--3--   --construct a property vector for high priority instructions--
--4--   }                    --end of instructions body--
--5--   BEGIN . . .          --here is the rest of the master--
```

If there are no low-priority instructions, lines 1 and 2 may be omitted. Interpress defines defaults in such a way that low-priority instructions are not often needed.

The need for the EXCH on line 2 of the example arises because of the way Interpress executes the instructions body. Before execution begins, the vector of external instructions is placed on the stack; the EXCH is exchanging the low-priority instructions with the external instructions. Thus, when the instructions body finishes execution, there are three vectors on the stack: the low-priority instructions, the external instructions, and the high-priority instructions (on top of the stack). These instructions are all merged together, with those defined in vectors closer to the top of the stack taking precedence over other definitions; hence the division into low and high priority instructions.

18.3 Standard practice

Creators should observe some standard conventions in order to insure orderly printing of the masters they create. These conventions merely require a master to specify a small number of printing instructions in every master.

A master should always give instructions for *breakPageFont, docName, plex,* and *media.* The last two need not be given if the creator is willing to accept the defaults specified by Interpress (a printer-dependent default for *plex* and 8½×11 inch plain paper for the media). If the *media* instruction is given, it is best if it is a low-priority instruction, so that an external instruction can override it. A master may optionally specify any other master instructions given above in Section 18.1.1.

Example 18.1 shows about the simplest master instructions. The default media is used.

```
--0--   {                    --beginning of instructions body--
-- --   --no low priority instructions--
--1--   breakPageFont        --property name--
--2--    xerox xc82-0-0 times 3 MAKEVEC    --value=font name--
--3--   docName              --property name--
--4--    <Introduction to Interpress> --value=string in breakPageFont--
--5--   4 MAKEVEC            --construct a property vector for high priority instructions--
--4--   }                    --end of instructions body--
--5--   BEGIN . . .          --here is the rest of the master--
```

Example 18.2 is similar, but requests 8½×14 inch paper (remember that coordinates are expressed in meters).

```
-- 0--   {                    --beginning of instructions body--
-- 1--   media                --property name--
-- 2--    default 0.2159 0.3556 3 MAKEVEC 1 MAKEVEC --value=vector of MediumDescription--
-- 3--   2 MAKEVEC            --construct low priority instructions--
-- 4--   EXCH                 --as per template--
-- 5--   breakPageFont        --property name--
-- 6--    xerox xc82-0-0 times 3 MAKEVEC    --value=font name--
-- 7--   docName              --property name--
-- 8--    <Introduction to Interpress> --value=string in breakPageFont--
-- 9--   4 MAKEVEC            --construct a property vector for high priority instructions--
--10--   }                    --end of instructions body--
--11--   BEGIN . . .          --here is the rest of the master--
```

19

Printer capabilities

Not all Interpress printers are required to interpret all the facilities of Interpress. Some printers may have image-producing hardware that cannot handle complex images; other printers may have software that can handle only some parts of Interpress. While the Interpress standard can encompass printers with few or many capabilities, a creator may choose to limit the Interpress masters it generates so as to be able to print on as wide a variety of devices as possible.

Limitations on a printer's capabilities must be anticipated by the creator when the master is generated. There are several dimensions along which a printer may be limited:

- The printer may interpret only a *subset* of the Interpress facilities, i.e., a subset of the operators and data types defined by the standard. Interpress defines several useful subsets.
- The printer's environment contains only a limited collection of fonts, forms, and so on. Moreover, the printer may not provide elaborate font-approximation facilities. A master may limit its use of the environment, for example by restricting the fonts it uses to those listed in the metric master as having *easy net* transformations.
- A printer may not be able to handle certain printing instructions (Section 18).
- A printer's hardware and software may impose limits on the total *complexity* of an image it can print. A master can reflect these limits: for example, it might place no more than 300 characters on a line, or more than 5000 characters on a page.

Note that these are all limitations of a printer, not of Interpress. In this section, we shall consider only the *subset* that defines a printer's capability and the problems of limiting image complexity.

The capabilities of a printer must be communicated in some way to the creator. There are two mechanisms to achieve this: descriptive documentation published by the printer manufacturer or by the printer installer, and the metric master, which lists the fonts available in the printer's environment. Future extensions to Interpress will include mechanisms for providing the creator more information about a printer's capabilities.

19.1 Subsets

Interpress defines *subsets* of the standard in order to describe a particular printer's capabilities (§ 5.1). While the notion of predefined subsets cannot capture all of the subtle distinctions that may crop up in printing hardware and software, they provide some general categories into which Interpress printers fall.

All Interpress printers are required to implement the *text* subset, which is described in detail in § 5.1.1. Text Interpress contains all data types and operators of the Interpress base language except those involving control, testing, or arithmetic (e.g., IF, EQ, ADD). The text subset contains all the text imaging operators and also the simplest graphical operators: only black ink, no pixel arrays, horizontal and vertical strokes, simple outlines such as rectangles, translation, rotation by multiples of 90 degrees, and scaling.

The text subset is the starting point for several possible *enhancements*, illustrated in Table 5.1 of the Standard. For example, the base language can be enhanced to include the *Computation* module, which contains the control, testing, and computation operators. The imaging operators can be enhanced along several dimensions independently. For example, a printer that can handle pixel arrays in addition to the text subset is said to possess the *Binary* enhancement. A printer can be characterized by the list of enhancements it supports. If it supports only the text subset, it provides no enhancements. A printer that supports the *Computation* and *Binary* enhancements can be described as a printer whose subset is "*Text* with *Computation* and *Binary*."

19.1.1 Limits

One of the dimensions of Interpress subsets is *limits*. The limits of a printer define the minimum sizes and precisions of various important objects that the master uses. The limits for the *Text* subset are given in Table 19.1. Note that these are minimum limits—a printer may exceed these, for example, it might use $maxInteger = 2^{32} - 1$.

Table 19.1 Limits

Name	Where defined	Minimum limit
maxInteger	§ 2.2.1	$2^{24} - 1$
maxIdLength	§ 2.2.2	100 characters
maxBodyLength	§ 2.2.5	10000 literals
maxStackLength	§ 2.3.1	1000 values
maxVecSize	§ 2.2.4	1000 elements
topFrameSize	§ 3.1	50 elements

19.1.2 Other resource limitations

When the definitions of Interpress subsets are too coarse to describe precisely the capabilities of a printer, additional descriptive prose must be added to the formal subset definition.

However, in no case may an Interpress implementation fall short of the functions and limits of the *text* subset.

Some of this additional information arises because the subsets are too coarse. For example, a printer might offer more than 50 elements in its top frame, reasoning that more frame elements are needed to hold fonts for complex printing jobs. In this case, the printer's capabilities might be phrased as "*Text* with *topFrameSize* limit of 100."

Other examples of the coarse nature of subsets are drawn from imaging operators. If a printer offers the *Text* subset, but is also willing to draw diagonal (45 degree) lines, this additional function must be specified explicitly. Some printers may be able to handle only certain kinds of pixel arrays, those which are transformed with a particular *easy net* transformation (§ 5.1.2 and Section 13.5). The description of the printer's capabilities will need to list the easy net transformations it can handle.

Perhaps the most common sort of additional information concerns resource limitations that go beyond those in Table 19.1. For example, a printer may limit the maximum size of an Interpress encoding because the disk storage available to it for buffering the master is limited. Or a printer might limit the maximum number of pages in a master or in a printing job in order to cope with various operational problems, such as clogging the printing queue with long jobs.

Generally, a printer's capabilities will be described crudely by a subset description, but additional detail will be furnished as prose.

19.1.3 Image complexity

Perhaps the most difficult problem in describing a printer's capabilities is that of overall image complexity. Can the printer produce an image of 50,000 "A" characters? Of 1,000 strokes? Can it print more strokes if the strokes are only horizontal and vertical and not diagonal? What if 100 characters are all overprinted directly on top of one another? These are questions that have no simple answers because the answers depend on details of printing hardware. In many cases, it is impossible to express the capacity of the printer with a formula that will be of use to the creator.

Some so-called *unlimited* printers can handle imagery of arbitrary complexity. These printers operate by storing a raster representation of the page image to be printed. An Interpress interpreter clears out the raster at the beginning of each page, after which each imaging operator makes the appropriate changes to this raster. The interpreter takes as much time as necessary to make all the required changes to the page image; complex pages require more time than simple pages. When the interpreter finishes the page, the stored raster is read out and sent to the printer hardware. This technique requires sufficient storage to save an entire page image and requires that the rate at which data are read from storage exceeds the data rate of the printing hardware. Printers of this design have the nice property that they will print arbitrary pages; simple pages are generated quickly and more complex pages more slowly.

High-speed printers avoid storing the raster image of the entire page by generating it "on-the-fly" as the printing hardware consumes it. While this technique has impressive performance advantages, it limits image complexity and leads to *limited* printers. If the printing hardware is consuming raster data at a fixed rate, a complex image may require more time to generate the

raster than is available. The printer grinds merrily onward, but the generation of the image data has fallen behind. Once the data-generation and printing processes lose synchronization like this, the rest of the image may be spoiled. The printed page is a mess!

Some printing hardware makes unlimited complexity easier to handle by allowing the printing to stop if the data-generation process has been slowed down by complexity. As you might imagine, however, starting and stopping a printing engine frequently is a difficult mechanical engineering task. Most high-speed printers won't allow it.

19.2 What a printer should tell you

A creator needs to know certain information about a printer in order to prepare masters for it effectively. The metric master contains some of this information: a list of fonts stored in a printer's environment and the metrics associated with them. However, additional information is sometimes required.

Printer information comes from two sources. Some of the information is determined by the printer hardware and software, and so is specified by the printer manufacturer. Other information may be specific to a particular printer installation, and so must be determined by the organization responsible for installing and operating the printer.

This section attempts to summarize the printer information that should be supplied by the manufacturer and by the installer.

19.2.1 Manufacturer's information

The following information is generally supplied by the manufacturer of an Interpress printer:

- *Subset.* What is the subset of the Interpress language that the printer can accept? Besides subset names, this information should include any prose required to specify the capabilities of the printer. For example, if only certain pixel arrays can be handled, acceptable easy net transformations should be listed. If the printer restricts resources not listed in the limits table (Table 19.1), these should be clearly identified. Any special performance hints, especially any that are counter to intuition, should be explained.
- *Image complexity.* Is the printer *limited* or *unlimited?* If it is limited, some attempt should be made to characterize the maximum page complexity that can be handled.
- *Image fidelity.* How precisely can the printer render the image specified in the Interpress master? In most cases, the resolution of the printing hardware will determine the answer to this question.
- *Font approximations.* How are font approximations made? The approximation rule used by the printer software should be described. In some cases, printers might adopt conventions about the interpretation of hierarchical font names, which should be described.
- *SequenceInsertFile.* What are the syntax and semantics of *sequenceInsertFile?*
- *Printing instructions.* What printing instructions are available? Are there any installation-dependent printing instructions?
- *Transport media and communications.* How does the creator communicate with the printer? This includes information about transmitting masters to the printer, determining the status

of a print request, and obtaining the metric master. It is especially important to describe the kinds of errors that can occur and how they are reported.

- *Environment.* What are the fixed elements of the printer's environment? Decompression operators are usually "built in" to the printing hardware, even though they are expressed as an environment object in Interpress. The function of these operators must be described precisely so that creators can compress pixel data accordingly. What colors are available? Some printers may also have a few fonts permanently resident in the environment, which should be described.

Still more printer information will be provided by the manufacturer, such as how to operate it, how to install forms on it, how to update fonts, and so on. This information is vital to the operator of the printer, but is not required by the creator software.

19.2.2 Installation information

Each printer installation may be configured differently. If printers have different fonts, the creator is informed via the metric master. However, other differences are important to the creator as well:

- *Environment.* What forms are available on the printer to be used with *sequenceInsertFile?* What are the hierarchical naming conventions used in the organization, and are there any guidelines about generating masters so as to achieve maximum device-independence on all the printers in the organization?
- *Printing instructions.* An installation may restrict some of the printer's capabilities. For example, it may stock only certain paper sizes, it may not allow stapling, or it may use special cost-accounting practices.
- *Operational restrictions.* For various reasons, the operator of a printer may restrict its use in certain ways. The maximum number of pages in a print job might be restricted, or restricted during certain peak hours in order to keep the printer queue flowing smoothly. A creator might be able to work around a length restriction, so will need to know about it.

Installation information could be included in a profile file that the creator reads when generating a master or when transmitting a master to a printer. In this way, an installation could change this information without having to reprogram Interpress creators.

20

Performance

Performance of printing systems is often very important. Large computer-printing installations have many high-speed printers, each printing one or two pages a second, working three shifts to print enormous volumes of reports, invoices, shipping forms, and so on. While the images on these pages may not be very complex, they must be printed quickly.

Interpress masters can describe simple documents of this sort, but can also define very complex high-quality images found in books, magazines, and journals. Preparing these more complex images for printing is a bigger job than making the computer-printing images and can be expected to require more computing resources.

Interpress accommodates these two performance extremes. For simple images, a master can be interpreted and printed quickly. If complex image-generation facilities of Interpress are used, interpretation and printing may slow down. For high-performance applications, a master should restrict its use of Interpress operators to those that are designed to be handled efficiently.

This section explains some of the considerations behind the performance of Interpress printers and how to create masters that will print efficiently.

20.1 Interpretation and printing

Most Interpress printers will use a two-step process to print a master. First, the master will be *interpreted* and some form of device-dependent intermediate representation of each page will be built. The time required for this step will vary depending on the complexity of the master and a great many other factors. In the second step, the intermediate form will be used to drive the printing device to *image* the document. For printing devices that operate at fixed rates, e.g., two pages per second, the time required to carry out the second step is proportional only to the number of pages printed and not to their complexity.

Some printers will perform these two steps serially, that is, the printer will either be interpreting or imaging. More capable printers will be able to overlap these two processes so that while one document is being imaged, another document is being interpreted. Some printers will interpret and image entire documents, while less capable printers may interpret and image on a page-by-page basis.

If several identical copies of a document are being printed, the master needs to be interpreted only once to create an intermediate representation that will be used repeatedly to make the necessary images. If copies differ due to the use of the IFCOPY operator, those pages that call IFCOPY may need to be interpreted once for each copy being printed in order to obtain the proper image for each copy.

20.2 Efficient masters

There are three principal ingredients to determining the efficiency of interpretation of an Interpress master. First, the kind of imaging operators used is important. Second, a set of reasonably intuitive observations about how the master is written apply. And third, as in any programming environment, some algorithms are more efficient than others. In addition to the considerations discussed here, Section 17.1 provides a number of hints that lead to efficient masters.

20.2.1 Imaging operator efficiency

Interpress masters that build only simple images will be interpreted more efficiently than those that use complex facilities. Interpress imaging functions can be broken into three rough classes:

1. Formatted multi-font text (Examples 3.2, 3.5, 3.7)
   - text positioned with SETXY and printed with SHOW
   - only easy character transformations (Section 13.5) are used
   - narrow horizontal and vertical strokes, or "rules"

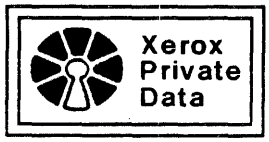
2. Line drawings. Class 1 plus:
   - arbitrary strokes

3. Complex images. Class 2 plus:
   - filled outlines
   - pixel arrays
   - colors other than black
   - priority
   - complex instancing
   - arbitrary character sizes and rotations

These classes are only suggestive; some printers may provide special features that make certain functions in Classes 2 or 3 just as efficient as those in Class 1. For example, a printer might support a certain kind of pixel array just as efficiently as text characters.

20.2.2 Interpretation efficiency

Programmers learn over time which programming constructs are efficient and which are slow. For example, Pascal programmers are often admonished: "Don't pass arrays by value unless you must, because the entire array is copied on each call." Some of these lessons are quite intuitive, such as that comparing two pointers for equality is faster than comparing two records for equality. Others may have no intuitive basis, but simply reflect implementation performance of the language's compiler or of the computer on which it runs.



The Interpress language has the same properties. Intuition and the descriptions in the Standard are generally helpful guides to the resulting interpretation speed. Perhaps the most important thing to remember is that interpretation speed depends in large measure on the number of tokens that must be interpreted. Thus, for example, using a *sequenceString* token to encode a vector of character codes will be far more efficient than the equivalent form that stacks each element of the vector and constructs a vector with MAKEVEC.

Performance is increased if the number of calls on primitive operators is decreased. Thus it is far more efficient to print a twenty-word line of text using a single call to SHOW than to use twenty calls to SHOW, one for each word.

Interpress encounters some overhead in calling composed operators, just as conventional programming languages often have significant procedure-call overheads. Thus composed operators are helpful in saving space in the master, and necessary in some contexts (e.g., IFCOPY's *testCopy* argument), but generally reduce interpretation speed.

20.2.3 Algorithm efficiency

Some algorithms are inherently more efficient than others. This observation applies to image-generation algorithms just as forcefully as it does to sorting algorithms. Although Interpress provides a number of ways of specifying identical images, the performance of the methods usually differs. A simple technique that costs dearly in performance and doesn't achieve much image-generation is writing "white" to obscure parts of an image that need never have been generated. Example 15.8, which uses this technique, is likely to be much slower than its equivalent that uses strokes, Example 3.1. While you might choose to view the difference in performance as a difference between the speed of Interpress operators, it is also a difference in basic image-generation algorithms.

20.2.4 Performance constraints

Some printers may choose to define constraints on Interpress masters so that they will execute efficiently. Such a printer must still implement all of the Interpress standard for a particular subset, but it need not implement all features equally efficiently. For example, a performance constraint might dictate that certain master coordinate systems will require less computation than others, perhaps because the coordinates correspond exactly to device-dependent coordinates. However, such a printer must still allow arbitrary master coordinate systems, subject only to the restrictions imposed by Interpress's subsets (§ 5.1.1).


Xerox
Private
Data

21

What can go wrong

When an Interpress master is printed, there are a number of things that can go wrong. There are two broad classes of problems: operational problems related to the particular printing request made, and problems with the Interpress master itself.

There are a number of operational problems that may crop up, such as a broken printer, a paper jam, incorrect routing of the printed document, and so on. The printer may reject a job because its supply of the right kind of paper is exhausted, because a data error is detected during the transmission of the master to the printer, or because the requestor has exceeded his alotted printing funds. Some of these operational problems are transitory, such as when the printer's job queue becomes full. In this case, the requestor can try again later.

Once past operational impediments, problems may arise because of the construction of the master. Interpress printers are designed so as to minimize the disastrous effects of errors in a master: in most cases, the interpretation of a master can continue without spoiling more than the one page where an error occurred. Some errors, however, such as those detected when the preamble is executed, may prevent printing any part of the document.

21.1 Errors

Printing errors are divided into two categories: appearance errors and master errors. Appearance errors are those that affect only the appearance of a page; in effect, these are errors detected by the imager. Master errors signal problems that arise as the master is decoded and interpreted; these errors are detected by the interpreter.

Errors are further divided to yield four categories:

- *Appearance warning.* An appearance warning signals that the image created by the printer will differ in a small way from the ideal image specified in the Interpress master, but the perceived *content* of the image will not be changed. For example, a font approximation generates an appearance warning.
- *Appearance error.* An appearance error indicates that the imager had to make an approximation to the ideal image represented in the master in such a way that the resulting image will not appear to be correct. For example, if an imager cannot print a pixel array mask that is specified in the master, an appearance error will result.

- *Master warning.* A master warning indicates that the interpreter has encountered an error whose severity is not likely to prevent the interpreter from continuing. For example, arithmetic overflow or division by zero cause master warnings.
- *Master error.* These errors signal problems so severe that the interpreter cannot continue to execute the master normally. For example, stack underflow and calling FGET with an out-of-range argument generate master errors. The interpreter takes explicit error-recovery measures to try to continue printing.

21.1.1 Error logging

An Interpress printer should provide diagnostic information about each error it encounters. This information can often be printed on the break page, but may be presented in other ways as well. An error indication should include, at a minimum:

- the page number;
- the class of the error (appearance warning, appearance error, master warning, or master error);
- some indication of the nature of the error.

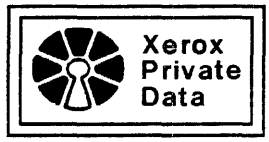
Printers should take care to be sure that the error listing is meaningful. Suppose, for example, that the master makes 503 calls to the SETGRAY operator and that 304 of them cause appearance errors because they specify a color other than black, which is the only color the printer can handle. Printing 304 error messages is not a good idea. Most users would be content with a single error message that indicated that the printer cannot handle gray, perhaps with a list of pages that will not be correctly printed. By contrast, each font approximation that is made should be clearly indicated in the error listing, giving the name of the requested font and the name of the one actually used.

A printer may provide more comprehensive diagnostic information for the benefit of a programmer who needs to track down the cause of the error. The information might include such things as the location in the master where the error occurred, the contents of the stack at the time, the name of the primitive operator that caused the error, etc. Since most users will not want such detailed information, the printer might provide it only if requested by a printing instruction.

21.1.2 Error recovery

In addition to logging an error, the printer must recover and continue interpreting the master if possible. In the case of appearance warnings, appearance errors, and master warnings, the interpreter can continue normally. In the case of master errors, the interpreter needs to take drastic action before resuming interpretation.

The MARK mechanism, explained briefly in Section 12.1.4 and fully in §§ 2.2.3, 2.4.1, and 2.4.6, is used to indicate points in the master where interpretation can resume normally after a serious error. The details of this mechanism are explained in § 2.4.1. Each page body is executed with mark protection, so that if a master error occurs within a page body, the remainder of the page body is not interpreted, but interpretation of the next page body can proceed without difficulty. Note that even if a page body is not interpreted completely, a page will be printed that contains any output that was successfully generated before the error



occurred. If a master error occurs while interpreting the preamble, the printer might not be able to generate any output besides a break page that indicates the error.

21.2 Examples of errors

General discussions of errors are useful and reassuring, but fail to give a feeling for all of the kinds of things that can go wrong. This section attempts to give a list of the most common types of errors. Each error is annotated with a code that indicates whether it is an operational error (O), an appearance warning (Aw), an appearance error (Ae), a master warning (Mw), or a master error (Me). Some printers may not classify these errors in the ways indicated by the codes.

21.2.1 Starting a printing job

- The printer's job queue is full (O).
- The printer encounters a data error while reading the master from the medium that was used to transmit it to the printer (O).
- The printer is not working (O).
- The printer is lacking necessary supplies, such as paper or staples (O).
- Insufficient funds are available in the account to be charged for the printing job (O).
- The printing job cannot be handled because of operational restrictions, such as limits on the maximum number of pages printed in a job (O).
- An error occurs when the printer tries to interpret the printing instructions that accompany the master (Me).

21.2.2 Printer capabilities

- The printer's interpreter is of the wrong *subset*, as indicated by invoking a primitive operator that is not implemented (Mw or Me).
- One of the printer's *limits* is exceeded, such as the maximum stack depth (Me).
- Some printer resource, not given explicitly as a *limit*, is exhausted (Me). For example, storage for composed operators and vectors might be exhausted.
- A printing instruction requests a capability that the printer does not have, such as stapling (O).
- The master requests a capability that the imager does not have, such as gray scale, or scaling a pixel array or a character using a transformation that is not among the easy net transformations of the printer (Ae).
- The printer is *limited*, and one or more of the pages specified in the master are so complex that the imaging-generation hardware cannot keep up with the printer mechanics (Ae).

21.2.3 Problems with the master

Master errors and warnings:

- The Interpress master is not well-formed (Me). That is, the header, BEGIN, END, and well-formed page bodies cannot be found.

- The Interpress master is not encoded properly (Me). For example, a primitive operator code or a sequence type may be out of range, or the rules about encoding body operators have not been followed.
- A reference to the environment generated by *sequenceInsertFile* or FINDDECOMPRESSOR cannot be honored (Mw).
- One of the arguments passed to a primitive operator is not of the correct type or in the correct range (Me). For example, <1 SHOW> and <−1 IGET> will generate master errors.
- The master "program" does not terminate (Me). Printers will usually take some steps to detect whether a master is looping, such as establishing a maximum interpretation time or detecting the absence of calls to imager operators that actually generate output.
- A computational error occurs (Mw). Division by zero and GETCP applied to poorly-conditioned transformations are examples.

Appearance errors and warnings:

- A font approximation is made (Aw).
- The master has been created in such a way that the image is greatly distorted in the presence of font approximations. *This causes no error beyond the appearance warning that accompanies a font approximation.*
- A font that is requested can be neither matched nor approximated (Ae).
- Part of the image lies off the page (Ae).

While these lists of errors may seem imposing, in practice many of the errors occur very infrequently. Not counting operational problems, the only errors that are likely to occur frequently are appearance warnings resulting from font approximations, and even these can be mostly avoided by keeping adequate font libraries with printers.

22

Interpress systems

Interpress masters can play the central role in many kinds of computerized systems for the handling of documents, pictures, or graphical information. While this report and the Interpress Standard both emphasize the application of Interpress to printing documents that are created by data- or word-processing software running on large computers or individual workstations, Interpress can be profitably used in many other ways.

This section deals with the application of Interpress to a wide variety of printing and image-preparation tasks. There are three principal parts to the discussion:

- Interpress is a common format for expressing the output of diverse image-preparation systems.
- Interpress allows heterogeneous printing environments to be constructed that serve a variety of printing needs.
- Interpress can be applied to a number of new or unconventional applications.

22.1 Interpress as a common output format

Interpress can be used as a common format to assemble parts of a document that are created with different composition or illustration systems. We have seen in Section 1.5 and Section 16 how Interpress masters can be manipulated in order to combine pages from several masters or to merge onto a single page information from several masters. These techniques allow many different creators to cooperate in the generation of a single document.

Usually Interpress is but one of several data formats that are essential to an application. Consider, for example, a document-editing or composition system that provides a way for an operator to edit a document interactively. A version of the document is displayed on the screen to provide the operator with visual feedback on the current format of the document. Between editing sessions, the system saves a digital *document representation* on some permanent medium such as a floppy disk. It is this representation that records the current state of the document and that the operator "calls up" when further editing is necessary. That document representation is not an Interpress master. Normally, it records the *content* of the document, with only minimal record of the *form* of the document. While various *formatting commands* might be embedded within the text, these commands are not like the graphical constructs of Interpress.

Thus, a command can say "start a new paragraph" but give no details of where on the page the paragraph should be placed, or how much to indent the paragraph, or how much space to leave between it and the previous paragraph. When the operator wishes to create a hard copy of the document, he invokes a program to convert the document representation into an Interpress master. The program might be called a *formatter* or *composition system* or *document compiler* or simply a *translator.* It uses the document's content and its format information to determine the physical form that the document should take; this physical form is then represented as an Interpress master. In some systems, the operator is unaware of the process that translates a document into Interpress form: he simply issues a "print" command, and goes to fetch a cup of coffee while the various conversion programs do their work.

This example illustrates that while Interpress masters provide the key interface between an image-creation process and an image-rendering process, applications usually require other data representations as well. Thus Interpress serves as a common format for all images destined to be printed, but the images are usually constructed by computer programs from other, non-Interpress, representations.

Figure 22.1 illustrates some of the different ways that material can be prepared as Interpress masters and then combined into one final document:

- Composition systems. A digital representation of the document is prepared with the aid of an interactive text editor or word-processor. The representation often includes embedded codes to control composition. When a hard copy is needed, the composition program generates an Interpress master from the document representation. Some composition systems are able to show on a display a close approximation of the final page, with text in the proper position and presented with fonts that are roughly equivalent to those that a printer will use. These are sometimes called "integrated composition systems," in which the distinction between the editor and the formatter is hidden from the user.
- Composition systems driven from data bases. Often documents are prepared directly from databases, such as price lists, stock quotations, airline timetables, or financial sheets. In one application, the entire body of laws for a state are organized into a database that can be used both for preparing hard copy and for interactive search and retrieval. Several dictionary and encyclopedia publishers maintain a similar database: it can be queried or updated, but it can also be used as input to a composition system that will typeset the reference book.
- Illustration system. An interactive program can be used to prepare line-art illustrations in much the same way a text editor is used to prepare text.
- Office Information System. Advanced systems such as the Xerox Star [18, 19, 21] integrate composition, record-keeping, and illustration into a single workstation. The user sees a collection of facilities, carefully designed to be operated easily and to work together. Although these integrated systems are already equipped to handle many applications together, their use of Interpress as an output medium makes them able to participate in a more diverse system.
- Document scanner. Photographs, drawings, or other documents can be scanned to obtain a digital facsimile record. A scanning system can provide a display to view the scanned result and interactive commands to size, crop, enhance, and position the image. The result is expressed in an Interpress master using pixel arrays.

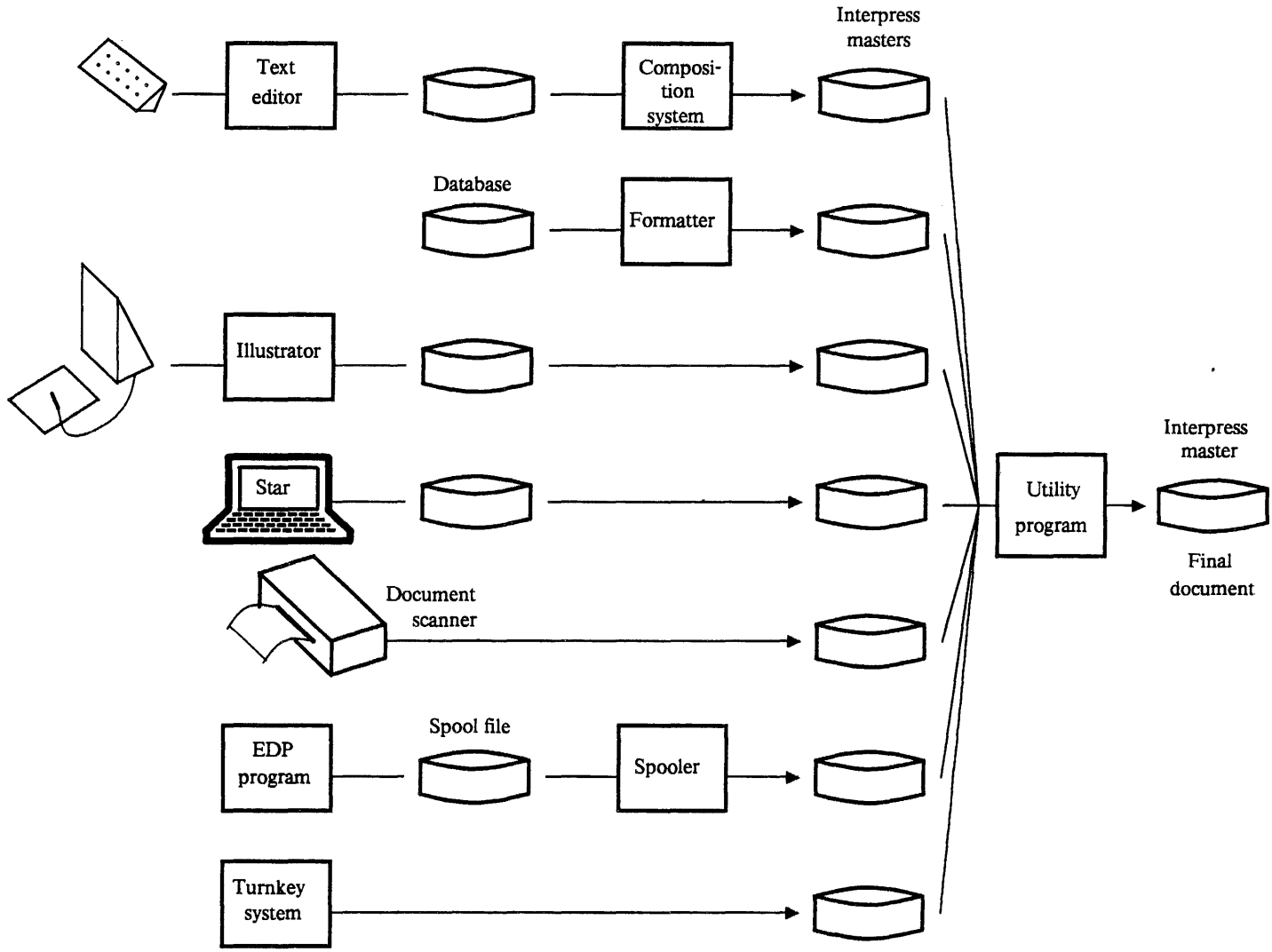


Figure 22.1. Interpress as a common output format.

- Data-processing programs. Most data-processing programs print some form of output, either a summary history of the program's execution and resource utilization or a larger "listing" that is one of the program's principal results. While these listings are often represented in a format designed originally for a line printer, a second program can convert these into Interpress masters.
- "Turnkey systems." A great many other systems fit into the same general model: they prepare descriptions of images that can be expressed in Interpress. For example, a computer-aided design system that builds and analyzes a model of an electronic circuit will have a command to make a logic drawing of the design. Typing this command will cause the program to prepare an Interpress master that can be printed.

Each of the systems in Figure 22.1 uses Interpress principally as a hardcopy output format: an Interpress master is prepared in order to control some kind of printing device.

The figure shows another function, however: all of the systems are contributing to a single document. Each one creates one or more Interpress masters, which utility programs merge appropriately into a single document. Because Interpress masters can be merged in this way, a wide variety of systems can be used in the preparation of a single document. The systems need not be located together, nor manufactured by the same company, nor even operated by the same organization. Old equipment can be retired and new equipment added to the system without introducing compatibility problems. All that is required is that the systems that contribute to the final document generate Interpress masters as output.

The Interpress master is the only common element necessary in this diverse set of document-preparation tools. This is a relatively weak requirement, which thus allows each participating system to be customized to its particular task. While it might be possible to integrate into a single system all the functions illustrated in Figure 22.1, the integration task would be difficult, and the system might be inflexible and hard to change.

In more traditional manual document preparation systems, paper or film are used to integrate images from several sources into one document. Each image-generation scheme, whether a typesetting machine or an artist with a brush, generates the common format of cuttable paper. The document assembly is then performed with knife and glue, yielding after suitable copying another piece of paper, which can be used as a final copy or used as input to yet another preparation process. By analogy, in a document preparation and publishing endeavor, Interpress serves as electronic paper.

22.2 Heterogeneous printing environments

Just as many different kinds of equipment can participate in the preparation of Interpress masters, many kinds of printing equipment can print masters. Different equipment may have different capabilities or speeds, or may be located in different places. If several printers are connected together with a computer communication network, they constitute a *heterogeneous printing environment* in which Interpress masters may be transmitted to and printed on any printer. Such a network is depicted in Figure 22.2.

The selection of a printer may depend on a number of factors, such as whether the printer is fast or slow, whether printing is expensive or cheap, how close the printer is to the place where the output is needed, what sizes or kinds of media are available on the printer, whether the equipment is operating or down for maintenance, whether the printer is capable of color or black and white, whether the printer is idle or backlogged, whether the printer's environment has the necessary fonts, and so on. A master might be sent to more than one printer in order to distribute copies to several geographic sites or to increase the printing rate of a large job by harnessing several printers at once.

While a heterogeneous printing environment offers a great many advantages, the geographic distribution of printers is probably the most noteworthy. Each floor of a large building or each branch office of a company might have a printer or printers handy to the people who work there. Printers might also be distributed by function, e.g., each of the several engineering services offices in a large company might have a printer to print engineering drawings prepared on computer-aided design equipment. A heterogeneous environment might be operated as a service business, with printers spread about many cities and courier service to deliver printed output. Such an environment might be connected to one or more public communication networks that clients could use to send masters to the printers.

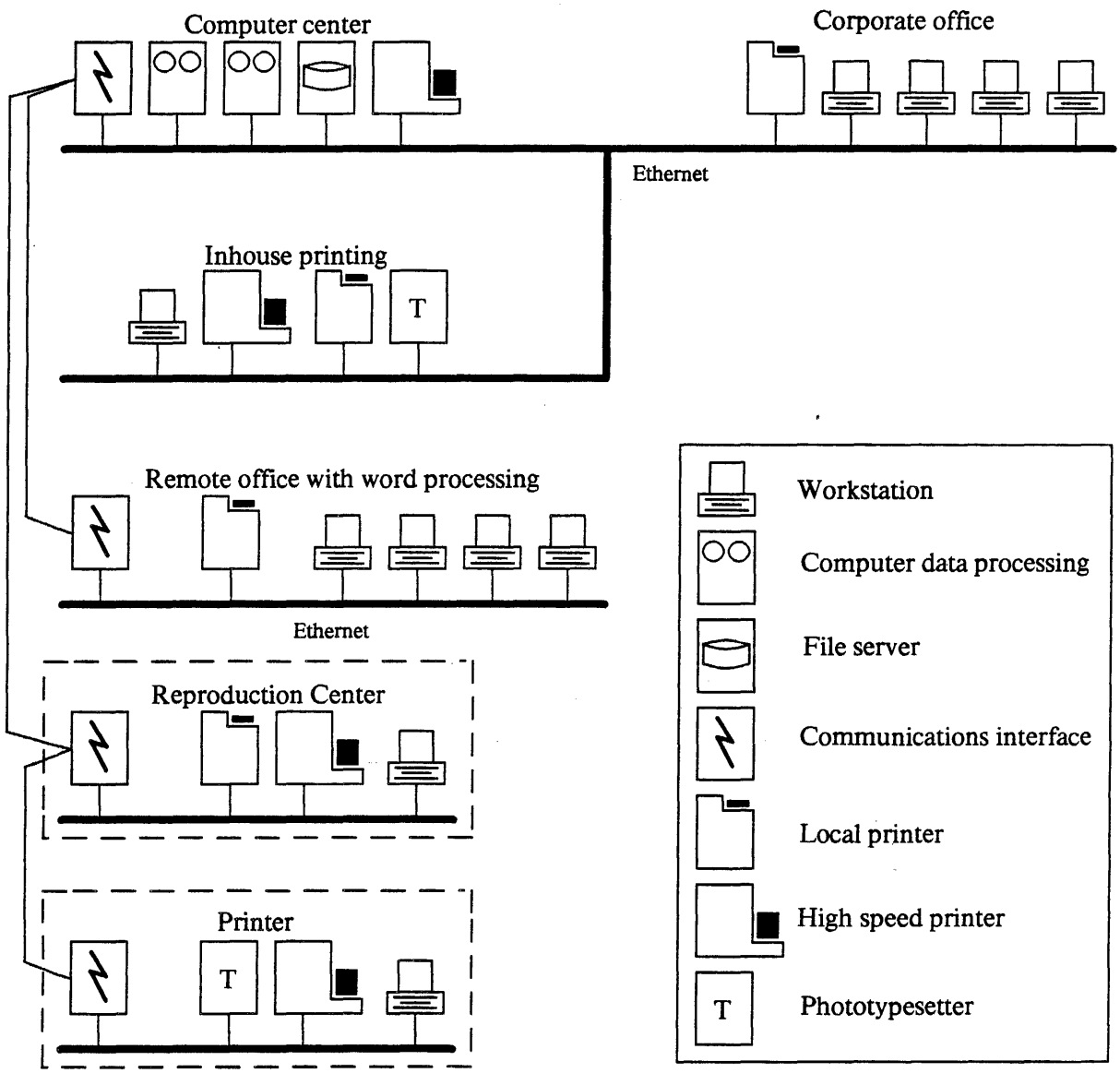


Figure 22.2. Heterogeneous printing environment.

22.3 Interpress applications

While the principal applications of Interpress have already been discussed, this section examines a few in more detail and shows how a standard electronic representation of a printable document can lead to new applications.

22.3.1 Digital interface to printing

Just as Interpress is invaluable for the creation and editing of documents—the so-called prepress operations—it is also very useful to an organization whose business is printing. An Interpress master and its printing instructions together constitute a complete digital specification of a print job. The scale of this job could range from a single copy of a one-page black-

and-white memo to several million copies of a bound full-color catalog. In both of these extreme cases, all of the information necessary to specify the job is recorded digitally: the master describes the printed images and the printing instructions provide the additional information such as how many copies to print.

Such a digital interface to printing allows the design and creation activities to be separated from the printing steps without danger of imperfect communication between the designer and the printer. The creation of Interpress masters can be undertaken by authors, illustrators, book designers, and publishers. Proof copies can be obtained by sending the master to a low-speed raster printing device accessible to the people creating the document. When the document is in final form, the master can be sent by a computer communication network to a printing plant, where the large-scale printing job is done.

The printer can treat the Interpress master in many different ways. If the job is a small one, the master can be sent to an Interpress laser printer that prints the job directly. Because laser printers are not practical for large jobs, the printer might prefer first to prepare images of each page on photographic film that can then be used to expose offset printing plates or to engrave gravure press cylinders. To prepare the film, the printer uses an "Interpress film printer," which is similar to a phototypesetter, but uses Interpress masters as its controlling input. With modern technology for exposing offset plates or engraving gravure cylinders directly from a digitized raster image, the Interpress film printer could prepare the printing plates directly.

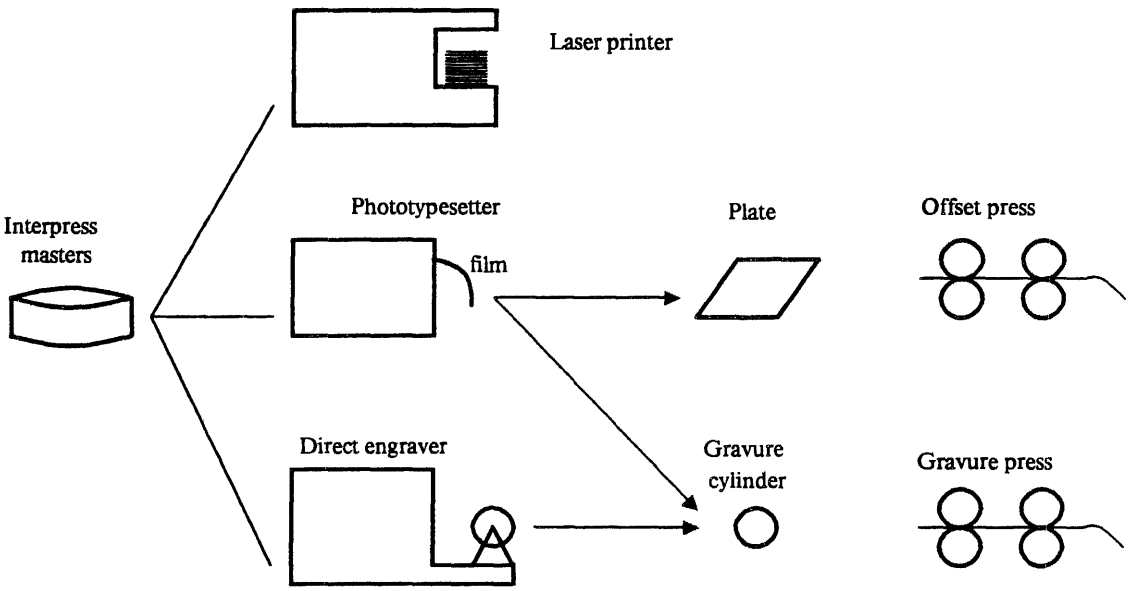


Figure 22.3. Some of the printing options.

While Interpress masters form the basis for a fully digital interface between a client and a printing plant, Interpress is also useful within the printing operation. The pre-press operations often performed by a printer are really the later stages of the document creation process: typesetting, layout, stripping in illustrations, etc. The systems illustrated in Figure 22.1 that perform these functions for document authors can be operated by a printer as well.

A printer can also use Interpress to help balance the load among a set of presses or Interpress printers. A central dispatcher can hold masters for all pending printing jobs and schedule them onto printing equipment according to the resources needed for the job (number of copies, type of binding, paper requirements, etc.) and the capabilities and availability of printing equipment. By using computer network communications, a printer with several plants could transmit masters to plants with idle capacity. Likewise, in the event of press breakdown, a master could be sent to another plant.

Interpress can revolutionize the nature of the communication between a printer and his customers. A customer desiring to purchase printing services from a printer can transmit his print job to the printer electronically, by copying the Interpress master and printing instructions over a network. A printer can give price quotations from examination of the printing instructions and a brief summary of the master, such as the number of pages and the level of Interpress that it requires. For customers whose technology or temperament guides them to want paper intermediate copies, a printer can offer digitally vended prepress service, such as typesetting of Interpress masters, with the result mailed to the customer or delivered like a pizza.

22.3.2 Demand printing

Interpress masters are an ideal document representation for demand printing applications. A large file system holds many Interpress masters, each one describing a printable document. A simple interactive management program is used to instruct the system to transmit a master to a printer so that one or more copies can be made. The system keeps track of which documents are printed by whom in order to bill customers and to pay royalties to authors of masters held in the file system.

By its very nature, demand printing allows documents to be printed only when they are needed, thus saving the cost of pre-printed but unused copies. Forms and form letters are ideal demand printing applications, since the forms can be updated without having to destroy a large inventory of pre-printed copies.

A demand-printing system is even more interesting when the file storage and printers are geographically separate and are connected by a computer communications network, as illustrated in Section 22.2. A customer could issue a command at a printer that indicates which document he would like to print. The printer would interrogate a file-storage system via the network; the master would be transmitted to the printer over the network, and finally the printer would make the requested copy. There is no technical reason why such an arrangement could not accommodate a large number of file-storage systems. An organization wishing to sell documents would place Interpress masters for them in a file store and connect the store to the appropriate computer network via some accounting software.

22.3.3 Information distribution

Interpress can be used to distribute hardcopy information electronically in a heterogeneous printing environment simply by sending masters to printers at appropriate sites. For example, large companies distribute daily news announcements to all their branches; research departments of stock brokerage houses distribute market news to account executives; national newspapers distribute made-up page masters to local printing plants; universities with remote

television instruction distribute copies of assignments and lecture notes to far-flung students; the Internal Revenue Service distributes tax form masters to computerized tax preparers. A network of printers that all obey the Interpress standard can easily perform such distribution.

22.3.4 Graphical information in query systems

Online database systems are an increasingly important part of modern business. We depend on databases of medical or legal information, library contents, stock market quotations, news stories, and more. In Europe, various telecommunication services are testing online query systems as a substitute for the familiar telephone book. Though some of these databases, such as the international news wires, have developed schemes such as "wirephoto" for including pictures with the text, these schemes are slow and wasteful of storage, and are not appropriate for general-purpose retrieval systems. Many installations now use microform readers with attached photographic printers; if a browser needs a printed copy of a page image, he can cause the image to be photographed on the page printer, and take the page away with him. While effective, these page printers are slow and expensive. The telephone information systems do not provide enough graphics capabilities for a proper yellow pages.

A database scheme based on Interpress would bring graphical information to online databases and reduce the price of local-origin copies such as those now done with microform page printers. Although the cost per page of microform representation is currently much lower than the cost per page of disk storage representation, archival read-only material suitable for microform storage can be stored in Interpress format on digital video disks, which are cheaper and more easily searched than microform.

22.3.5 Computer-assisted instruction

Computers are increasingly valuable as teaching devices; each student can progress at his own pace through a programmed instruction script, with the computer providing information, interaction, stimulus, and instantaneous correction. The vast majority of these CAI systems are entirely textual, and those that do provide graphical instruction capabilities do so at great expense of preparation and storage, and no two schemes are quite compatible.

By representing a CAI script as a series of Interpress page images embedded in the logical framework of the lessons, a script writer can create an illustrated lesson with no more work than it would take to prepare the same lesson in a more traditional textbook. The student might then have the option of printing a lesson and studying at home or displaying pages on an interactive terminal at school.

22.3.6 Engineering data distribution

Many high-technology component firms, such as those selling integrated circuits, spend a great deal of effort preparing and distributing "data handbooks," which are collections of product information sheets for these company's product lines. Each product information sheet typically contains some textual description of the capabilities of the product, but also diagrams, timing charts, performance graphs, and other pictorial information. Since these data handbooks become rapidly obsolete as the product line changes, it would be beneficial to component vendor and design engineer alike if the data handbook could be stored online on the design engineer's computer, and updates transmitted electronically. By using Interpress to represent the page

images of the product information sheets, new sheets can be distributed rapidly and efficiently to interested engineers. Once the electronic data distribution network is in place, suppliers could include as part of their online product information sheets the component drawings used by electronic computer-aided design systems.

22.3.7 Projection images

Interpress can be used to represent the text and graphics that are projected on a screen as part of a technical or sales presentation. An Interpress master drives an imager that creates an image on a television display, which can be projected on a large screen. The speaker can control the rate at which pages are displayed and which page to display next.

Interpress can also be used to transmit presentation material to remote imagers. In this respect, it serves a role similar to a "presentation standard" such as Videotex [17]. The images may be generated by an interactive program or may simply be retrieved from a database.

Interpress can also be used to produce colored slides for a presentation. A business-graphics or illustration would prepare the appropriate Interpress master, which could be transmitted to a high-resolution film recorder equipped with Interpress software. Such a film recorder might be operated as a service accessed by the fully digital interface Interpress offers.

22.3.8 Spooling and computer printing

One of the intended uses of Interpress is to provide computer printing services to batch or time-sharing computer systems. One or more Interpress printers are connected, perhaps using an Ethernet, to the computer. The application programs and operating system on the computer cooperate to prepare and send masters to the printer.

The early days of data-processing gave rise to a technique called *spooling* for dealing with printer output. Spooling solves a problem that arose when operating systems began to allow more than one program to work concurrently, either by time-sharing the CPU or by the corresponding technique for batch programs, called multiprogramming. The problem is that although the computer has only a single line printer, two or more of the running programs might be creating line-printer output. Only one program gains access to the printer, thus delaying the other one. An additional problem is that some programs do a great deal more computing than printing, so only one line a minute might be printed. The sad result is that several programs are delayed because only one can use the printer at a time and because the one that's printing is slow.

Spooling decouples a program's execution from the actual printing process. The idea of spooling is to write a program's output on a disk file, even though the program thinks it's being printed. Since it is assumed that there's always sufficient disk space to hold these spool files, several programs are allowed to execute concurrently, each one thinking it has access to "the printer." When a program instructs the operating system that it is finished with the printer, the operating system adds the spool file to a queue of files to be printed. This queue is examined by a program called "the spooler" that runs more or less all the time to drive the printing device at full speed, sending information from the spool files to the printer.

The spooling technique can also be applied to driving Interpress printers. Some printers may have communications interfaces that allow only a single master to be received at a time. Other

printers, especially inexpensive ones, may not be able to initiate printing activity until all communication activities have completed. Both of these possibilities suggest that spooling can help improve the printer's performance because spooling allows only a single master to be transmitted at a time even though several programs may be generating Interpress masters that are being recorded on spool files. Spooling also increases the transmission speed because the spooler can transmit files far faster than a typical composition system can create them.

Spooling also solves another problem that arises when Interpress is fitted onto a computer system, namely converting "text" files into Interpress masters. Most computer application programs that generate printed output do not prepare Interpress masters; instead, they write "text" to the spool file, often using some convention to convey formatting information, e.g., carriage-control characters, line-ending characters, line blocking, and so on. In most cases, it is not feasible to modify these programs to generate Interpress masters instead of line-printer formats. However, the spooler program can generate an Interpress master "on the fly" as it transmits the spool file to the Interpress printer. As described in Section 9.6.1, the spooler can examine a user profile to select appropriate fonts and to obtain formatting instructions. Alternatively, this information can accompany each entry in the spooler's queue.

Ideally, the spooler should allow both text files and Interpress masters to be entered in its queue. This facility caters both for unmodified programs that generate text, or "line printer" output, and for programs such as composition and graphics systems that generate Interpress masters.

23

The design of Interpress

This section explores some of the considerations behind the design of Interpress. Because the Interpress design takes a rather unconventional approach to the problem of controlling digital printing systems, it is important to understand some of the reasoning behind the design. This section sketches the background and design criteria for Interpress but does not attempt a detailed explanation of the reasons for all aspects of the design.

23.1 Background

Interpress is the culmination of ten years of research by the Xerox Palo Alto Research Center (PARC) into digital printing technology. Research has been directed both at the problems of creating, exposing and printing raster images on xerographic printers and at the problem of designing the "print file format," the information that an application program sends to the printer. Interpress also builds on Xerox's experience in designing computer printing hardware such as the 9700, 8700, and 5700 computer printers. The summary we present here stresses work on the print file formats, not on the hardware and software technologies of digital printing.

23.1.1 A "listing" system

An early experimental system connected to a facsimile printer produced "listings" of ASCII text files sent to it over a local network. The objective of the system was to print program listings for computer programmers and to print simple memoranda and text documents. Soon, however, there was pressure to take advantage of the raster-scanned nature of the printer to print more exotic documents. People designed several different fonts for the printer, the file format was extended to include special control character sequences to select fonts, and the printing software was changed to store fonts on the system disk and to honor font-selection commands. Someone else wanted to print line drawings and pressured the system's maintainer into adding "vector" commands to the input file, again encoded using control character sequences. Several experiments were done to explore printing arbitrary raster patterns on the device.

While this system served well for its original objective, it demonstrated a critical problem. As formatting ambitions rose, people started objecting to various decisions made within the print-

ing software. Examples of contentious decisions were: the number of lines on a page, the handling of tab characters, the format of a page heading and page number, the positioning of sub- and super-scripts, and the calculation of inter-line spacing. Eventually, many of these decisions were controlled by an elaborate set of control character sequences embedded in the text file. The system's maintainer was constantly beset by requests for new features or control over internal formatting decisions. The clear lesson was: it's a mistake for printing software to make formatting decisions because different users and different applications have different needs.

23.1.2 A device-dependent system

The next experimental system was designed so that the printing software made no formatting decisions: the creator was required to send to the printer a file in a device-dependent form that could simply be sent to the image-generation hardware on the printer. This file contained data for each character of each font that was required, compressed in the form required by the printing hardware. All of the constraints of the printing hardware had to be known to the creation software and reflected in the file, e.g., the maximum size of font memory, the maximum number of image-generation commands on a page, and the specific order in which image-generation commands had to be sorted.

This strategy successfully excised formatting decisions from the printing system, but placed an intolerable burden on creators. The first problem was that each creator had to maintain a font library, since the printer did not. These files took up an unreasonable amount of space on small workstation computers; there was considerable waste in the hundreds of copies of a font, one on each workstation's disk; and issuing new or updated font libraries was a significant problem. A second major problem was that each application program that created printed output had to contain software that embodied a great many messy details of the printing hardware. When the hardware was changed or new constraints were imposed, the software had to change. But perhaps the most obvious difficulty was that new printer designs would not necessarily have the same properties as this one, and that new creator software would have to be written each time a new experimental printer was developed.

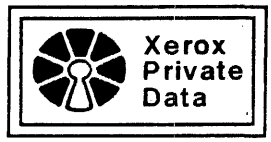
23.1.3 Press

In 1974, the experience collected from these initial experiments was used to design a device-independent print file format named *Press*. The principal objectives of this design were:

- The printer should make no formatting decisions (lesson from the first system).
- The printer should maintain a font library (lesson from the second system).
- The print file transmitted to the printer should be device-independent (lesson from the second system).
- The print file should accommodate both text and graphics. This need was becoming increasingly apparent as experimental document scanners and illustration systems were built.

Press is the direct predecessor of Interpress. Rather than explain the design of Press, it's easier to describe the differences between Press and Interpress:

- A Press file is a data structure, not a program. A separate descriptor and data format are provided for each kind of data: a string of character codes to print, a rectangle, an object to be filled, etc. No provisions are made for symbols or instances.



- All coordinates are expressed in Press in a single coordinate system, which uses units of 10^{-5} meters. Coordinates range between 0 and 32 cm.
- Press has no facility corresponding to CORRECT. If font approximations are made, text may be displaced.
- Press' graphics facilities include only filled objects and pixel arrays. The effect of strokes must be obtained by creating appropriate filled objects.
- Press was designed so that a Press file could contain both printing information and application information. The idea was that a Press file would be the only representation of a document, and that one could build interactive document editors that would edit a Press file directly. Such an editor would change both the document representation and the formatting information in the file.

Press proved to be extremely successful. It has been in constant use since 1974 and has been used to represent and print over 200,000 documents. The complexity of these documents has ranged from program listings to illustrated books. Software and hardware to print Press files have been implemented for over a dozen different printer designs. Numerous application and utility programs have been written that create Press files. To a very great extent, printer dependencies have been eliminated from this software altogether.

The lessons learned from Press are complex and not as easily summarized as those learned from earlier systems. Perhaps the biggest failure of Press was that the objective of recording in one file both a document representation and the print file did not work out—hard copies are produced sufficiently infrequently that the computing time and file storage required to keep the print information and document representation in synchrony were not justified. The list above of differences between Press and Interpress suggests other areas where the Interpress design has improved upon Press. More details about the reasons behind these changes are presented in the next section.

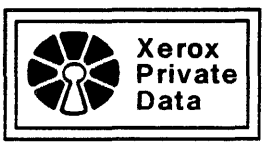
23.2 The design of Interpress

This section sketches briefly the rationale behind the design of Interpress. Because the design is rather intricate, this section cannot explain all of the considerations that went into it.

23.2.1 The language

Why is Interpress defined as a programming language in which calls are made to imaging operators that prepare an image? Why not instead simply define a data structure akin to a graphical display file?

The first observation to make on this issue is that the distinction between a program and a data structure is not a sharp one. A display file, while viewed as a data structure by the application program that creates it, is viewed as a program by a "display processor" that executes it in order to generate a display. The distinction comes down to one of the *instruction set* used: a graphical display file has instructions that correspond to graphical objects to be displayed; usually these instructions contain coordinate information as literal data embedded within the instruction. By contrast, the Interpress instruction set more closely resembles that of a general-purpose computer: it is defined without any particular reference to graphical objects, but creates images by calling appropriate procedures in a graphics package, which in Interpress is called the imager.



The approach of using a general-purpose language to control image-generation operators was chosen for several reasons:

1. A language is a more powerful framework for addressing device independence than is a graphical data structure. While a data structure is interpreted by a fixed image-generation algorithm in the printer, a language allows a creator to send to the printer an *algorithm* for achieving a particular effect on the printer. This algorithm can examine the printer's environment and compute an appropriate image. This property was illustrated in Section 17.3.1 by a master that builds a transformation so as to center the image on the output medium. To achieve carefully-controlled effects on a printer in a device-independent way, a creator can send an imaging algorithm encoded in an Interpress master. While a data structure can represent only one solution to an imaging problem, an algorithm can create an appropriate image, since it can sense important properties of the printer.

2. The language solves a problem that Press revealed, namely that some applications require very high coordinate precision while others require only modest precision. To allow variable precision with a data-structure approach such as in Press, we would have to define different instructions and instruction formats for coordinate data of different precision. The Interpress language solves this problem by making the precision of a literal a matter for the encoding of the command that stacks the literal value—the imaging operator itself is unaffected.

   One could argue that a similar technique could be applied to the data-structure approach by making all coordinate data self-describing, much the way Interpress token formats for encoding Numbers are labelled with their type.

3. A language provides a way to construct an operand that has many different forms. The best example in Interpress is probably a transformation. Suppose that the Interpress font setup template:

name FINDFONT *size* SCALE MODIFYFONT *frameIndex* FSET

was expressed instead in a data-structure form:

DECLAREFONT *fontNumber name transformation*

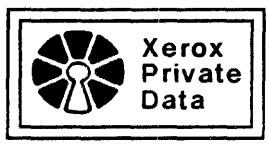
How are transformations of different types to be specified? Do we need many variants of DECLAREFONT, one corresponding to each distinct sequence of SCALE, ROTATE, and SCALE2 transformations, e.g.:

DECLAREFONT-RS *fontNumber name rotationAngle scaleFactor*

While there are other solutions to this problem, they all begin to take on the aura of a programming language.

4. A language that includes procedure-call instructions is a natural way to provide graphical symbols (procedure definitions) and instances (procedure invocations). Indeed, the most elegant attempts to design graphical languages have recognized this advantage [14]. Again, symbols and instances can be provided by more limited facilities in a graphical data structure. These facilities usually insist that instances cannot take arguments; thus, for example, a symbol cannot be defined to display a page heading because the page number cannot be passed as an argument. Interpress's composed operators not only allow instances to take arguments but can be used to carry out arbitrary computations.

5. Some people argue that the distinction between the data-structure and programming language approaches is that the programming approach provides a *store* operator that saves a



data value—FSET and ISET in Interpress. Even this distinction is not always clear. What is the difference between the Interpress font setup template and the data-structure form shown above? Both "store" a font description in a place that can be referenced later. A more interesting problem is posed by the imager variables, which represent, in effect, additional arguments to each imaging operator. These variables must be set, and often saved and restored. A general-purpose language provides a simple way to achieve these effects.

Expressing Interpress masters in a simple programming language costs very little. Most masters will be only a few percent larger than an equivalent data structure would be. It is likely that interpreting the program form will be slightly faster than interpreting the data-structure form.

While the Interpress base language bears a strong resemblance to a simple programming language, its facilities for preventing side effects stand out. The variants of DO that save various pieces of imager state were incorporated to deal with a number of problems that crop up in the printing application:

- Requiring the execution of each page body to be independent of any other so that the printer can execute page bodies in whatever order is necessary.
- Calling a composed operator defined externally, in the environment, while protecting the state that the master counts on, such as the contents of the frame and the current transformation.
- Preventing IFCOPY bodies from having side effects that alter the appearance of any part of the page not printed by operators contained within the IFCOPY body.

The absence of global variables and the simple mechanism of saving and restoring certain imager variables with DOSAVE and DOSAVEALL achieve all of these objectives.

23.2.2 Device independence

Perhaps the principal goal of Interpress is that of device independence. The experience with Press demonstrated that device independence was possible, but also revealed some problems.

The most important ingredient in achieving device independence is to express the desired result in sufficiently high-level terms that a printer can determine how best to achieve the desired effect. The master expresses the *objective* from which the printer must be able to determine the best *mechanism*. The choice of imaging primitives is the central issue. For example, suppose Interpress had no notion of stroke, but relied instead on the master defining strokes as outlines to be filled in. A printer can easily scan-convert the outline, but the quantization errors that are unavoidable in scan-conversion may lead to width variations in strokes. Even though the strokes are defined in the master to have the same width, the printer doesn't know that it should use a scan-conversion scheme that makes small position errors rather than width errors. By contrast, if the master contains a stroke description, the printer knows that preserving uniform width is an important objective.

Another important aspect of device-independence is the precision with which the Standard is specified. The detail in the reference document's definition of the Interpress language and arithmetic, for example, is sometimes overbearing. This precision is necessary, however, to insure that a master will be interpreted by two printers in the same way. While many "standard" programming languages avoid tackling arithmetic precision altogether, Interpress must specify the amount of arithmetic precision that is guaranteed by all printers.

The experience with Press uncovered a number of problems that a font library poses for device-independence. Press's use of a single 20-character font name showed the need for more flexible hierarchical names in Interpress. Also because of font naming conventions, Press had difficulty selecting good font approximations when it didn't find an exact match. Three other annoying problems uncovered by Press are solved by the CORRECT operator:

- Whenever a new version of a font is released, character widths may change slightly. When printer is sent a master that was created using old character widths, the lengths of text lines change. In Interpress, the proper use of CORRECT adjusts lines to have the desired length even when character widths change.
- When a font approximation is made, the widths of characters in the actual font differ from those in the desired font, and line lengths change. CORRECT helps preserve the appearance of a document in the presence of font approximation. Font approximation happens fairly frequently (a few percent of documents) for two reasons. First, if a font is deleted from a printer's environment, a master created and stored before the font disappeared will require font approximation when it is printed. Second, in a networking environment with thousands of workstations and tens of printers, it is not unusual for a printer to receive a master that requests a font available on printers in one part of the environment but not available on the receiving printer.
- When a master is printed on a low-resolution raster device such as a display, an especially difficult problem arises. The low resolution forces the printer to use device-dependent character definitions in order to achieve acceptably legible text. The low resolution of the device usually requires that character widths differ from the device-independent widths provided in metric masters and used as a basis for formatting masters (Section 9.6.3). Again, CORRECT helps deal with the problem by adjusting spacing to preserve line lengths when necessary.

23.2.3 Graphical primitives

The choice of graphical primitives in Interpress is determined in part by the capabilities of raster-scanned imaging systems, in part by device-independence considerations, and in part by practical considerations such as the size of the master.

While the natural form of image to present to a raster-scanned printer is a raster image, or pixel array, it is not reasonable to design a print-file format around raster images for several reasons. This reasoning is explained in some detail in Section 1.2.1; it is summarized here. First, a high-resolution image requires a great many bits to represent, even when sophisticated compression algorithms are used. Second, every creator would be burdened with the task of preparing a raster image, including maintaining a font library and executing scan-conversion algorithms. While special-purpose hardware could speed up image-generation, it's not as practical to make this hardware available to every creator as it is to package it in every printer. Third, and perhaps most important, a raster image has poor device independence. To be device-independent, a printer must accept a raster image of arbitrary resolution and convert it into an image whose resolution is appropriate for the printer. This computation is not only lengthy but usually introduces errors that are visually annoying.

All of these problems are solved by using a collection of "higher level" representations of graphical objects: strokes, filled objects, and characters. Pixel arrays are used as a last resort, when the image simply has no higher-level structure and can be represented only as a raster of

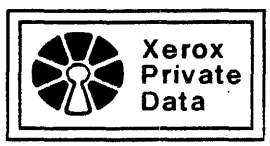


image samples. The higher-level objects allow compact masters, place little or no image-generation burden on the creator, and give the printer considerable room to maneuver to achieve device independence—we showed earlier that although strokes could be represented as filled objects, better device-independence is obtained by making strokes distinct from filled objects.

The printer also has considerable flexibilty in deciding how to form images of character shapes. Although the standard shows how characters *may* be defined as composed operators, they *need not* be defined that way. Here again, the master specifies the objective, namely to place an instance of a character according to a particular transformation. The printer knows the transformation and knows which character is called for, because it supplied a definition of the character. These two pieces of information give the printer complete information about the objective the master is trying to achieve. The printer chooses an imaging mechanism based on this information. For example, if the transformation calls for a common size and rotation, the printer may retain in its library a raster representation of the character, perhaps tuned to improve its legibility on the printed page. On the other hand, if the transformation is non-standard, the printer may retrieve an outline representation of the character from its library and apply the usual transformation and scan-conversion processes to make the desired image.

Transformations are provided in Interpress to solve a number of problems. First, they are essential for symbols and instances, which are very valuable image-generation tools. Second, they allow masters to specify images of arbitrary scale without necessarily requiring high precision. For example, using an appropriate transformation, a master that describes a billboard advertisement can use integers in the range 0 to 20,000 to denote positions measured to a millimeter over a range of 20 meters. If Interpress were to insist that all coordinates be given in units of 10^{-5} meters, the billboard would require numbers in the range 0 to 20,000,000, even though positioning precision of 10^{-5} meter is not required. Finally, it turns out that transformations do not slow down an Interpress printer because even a standard fixed coordinate system requires the printer to apply a transformation to convert to device coordinates.

Perhaps the most controversial aspect of the graphical facilities is the imaging model. Why not make it more general, so that the master could create output images and modify them? For example, why not allow a region of the output image to be "inverted," interchanging black and white regions, so that black-on-white text becomes white-on-black text? Why not allow arbitrary reading and writing of the raster image that is being prepared as output, to provide functions like BitBlt [9, 15]? Why not implement more general image-processing primitives?

One reason the imaging model is kept simple is to promote device-independence. The Standard allows printers to have very different representations for the page image, since no Interpress operators reference it directly or depend on the details of its representation. Color printers are especially likely to use different representations.

The Interpress imaging model is also kept simple to achieve good performance. The model represents a compromise between a fully-general image-preparation facility and something that printing hardware can be expected to implement in real time. An important overall objective of Interpress is that computer-printing devices can print one or two pages per second without requiring unreasonably complex image-processing hardware. Of course, not all Interpress masters can be processed this fast: those that contain a great many objects to be scan-converted or pixel arrays defined with a resolution different from the printer will require substantial processing time. But Interpress must insure that common cases can be processed quickly, which leads to a simple imaging model.

23.3 Relationship to other standards

Interpress is unlike other standards that impinge on digital printing in that it tries to represent the ideal image to be printed. Because Interpress attempts to codify a rather diverse set of images, it is related to a wide variety of standardization efforts. This section explains briefly these relations.

This discussion is worthwhile for two reasons. First, it is interesting to observe how the Interpress standard fits into other standards efforts. And second, it is important to understand whether a document represented using some other standard can be converted into an Interpress master and vice-versa.

23.3.1 Character set standards

There is a wide variety of character-set standards developed or under development. The recent word-processing and communication explosions have contributed energy to the work.

Interpress's notion of a *character* appearing on a page does not make reference to a character set standard. Rather, Interpress allows fonts to be given a name by the creator of an Interpress master; this name is interpreted by the printer to retrieve some information about how each character in the font should be printed. Although the collections of characters in fonts may correspond to a character set standard, Interpress does not require them to do so. Section 9.2 explains these considerations fully.

As character-set standards appear, Interpress can accommodate them. If both creators and printers agree to use a standard character set, the problems of interchanging Interpress masters between very different environments will be greatly simplified. It is premature, however, to recommend a particular character-set standard to serve in this way. Following are some activities underway:

ISO 6937. The International Standards Organization has prepared an extension of ISO 646 to include most characters required to print text in any language using a Latin alphabet. Unfortunately, it omits standard graphics required for high-quality printing, such as em dashes and en dashes, and fails to distinguish between hyphen and a "minus sign," a must for quality text.

Videotex/teletex standards. World-wide standards for presentation of text and graphics on consumer televisions are being developed. The International Telecommunications Union has named all such standards "videotex." Character sets are being developed in conjunction with ISO 6937, but with extensions to non-Latin alphabets such as Cyrillic, Greek, Hebrew, and Arabic.

ANPA. American Newspaper Publishers Association. This organization has embarked on the creation of a character set standard for use by U.S. newspapers. It is a small set, with many choices dictated by present newspaper requirements (e.g., bridge suit symbols).

Xerox. Xerox has developed an extensive character-set standard for use with its office-information systems [25, 29]. This set incorporates a number of other standards, such as ISO 6937 and standards for Greek, Cyrillic, and Japanese. The standard includes both *character codes* and *rendering codes* (e.g., for ligatures and other typographical artifacts).

Because Interpress imposes no requirements on character sets, it is trivially compatible with most character set standards. As remarked in Section 9.3.4, standards that use sequences of two or more codes to identify a single character cannot be accommodated easily in the Interpress framework. Since an ordinary "text file" cannot be sent to an Interpress printer in any case, it's a trivial matter to perform whatever character-set translations are necessary as the Interpress master is being prepared.

23.3.2 Page image formats

While character sets specify *how* each character should appear (its graphic image), page image formats (PIF) seek to specify formatting—*where* each character should appear. Compared to Interpress's formatting precision, the current proposed PIF's are crude, using monospaced fonts and concepts such as *line feed, carriage return,* or *backspace* that are held over from mechanically-operated printing equipment. Standards currently under development are:

ISO 6937. A portion of this document deals with "control functions for document interchange."

ANSI draft proposed standard for text information in page image format. Similar to ISO 6937.

Any of these PIF's can be converted into an Interpress master, because Interpress is able to express all of the formatting functions these standards require. The reverse, however, is not true: because an Interpress master can specify the positioning and appearance of characters very precisely, in general an Interpress master cannot be converted into a PIF.

23.3.3 Facsimile standards

Interpress can interoperate easily with facsimile standards, because facsimile encodings are based on raster-scanned imagery, which Interpress can accommodate. A facsimile encoding generated by scanning a document can be converted into an Interpress master in which each page is represented as a pixel array mask. Moreover, it is possible to convert Interpress masters into facsimile form, since the process of printing an Interpress master creates a raster-scan image of each page that can be designed to conform to resolution and scan-order conventions of any facsimile standard.

Much of the effort in facsimile standardization concerns data compression. In Interpress, data compression is left as an encoding decision, and indeed, some encodings of Interpress scanned images might use precisely the same compression techniques currently used by facsimile standards. (For a description of current facsimile coding work, see Hunter and Robinson [8].) As with character sets, Interpress can accommodate one or more compression standards by including in its environment a decompression operator corresponding to each standard (see Section 15.3).

23.3.4 Phototypesetter standards

At present, there are no existing or proposed standards for controlling phototypesetters. The Graphic Communication Computer Association (GCCA) is preparing a proposal for *generic coding* of text, which is derived from IBM's GML (*generalized markup language*). This standard applies to embedding formatting commands in text to be presented as input to composi-

tion software, not to the resulting composed pages. Thus it has no bearing on Interpress, which represents already-composed pages.

23.3.5 Computer graphics standards

Several efforts are underway to standardize software graphics packages used to build interactive computer graphics applications. The most prominent examples of such standards are the ACM-SIGGRAPH Core System [23, 7] and the GKS (Graphics Kernel System) [11]. These standards differ from Interpress in that they are subroutine packages to be used in a graphics application program and that they are designed to be interactive.

Any image created by a graphics package that implements one of these standards can be represented as an Interpress master. The image-generation primitives in both graphics standards can be converted into the Interpress representations for strokes and characters. Extensions to the Core for handling filled objects on raster-scanned displays will map to similar features in Interpress. Thus, Interpress can be used as a hardcopy output format for a graphics package.

In general, an Interpress master cannot be displayed using routines of these standards as the interface to an output device, because the graphics standards do not contain sufficiently versatile text- and raster-oriented functions. However, the packages can produce approximate images of Interpress masters that use only characters and thin strokes (lines).

A standard for describing static graphic images, more akin to the goals of Interpress, is the Virtual Device Metafile (ANSI X3H33). The standard specifies a file format to describe images containing lines, text, polygons, and scanned images. While the standard is quite complete, it does not offer as precise control over the image as Interpress; the difference is especially evident in the handling of fonts.

23.3.6 Presentation standards

The emergence of low-cost telecommunications and cable television has given rise to a number of standards designed to describe images that can be presented on home televisions equipped with a small amount of image-generation hardware. Numerous data-retrieval services can use these standards to display to a customer such things as advertisements, catalog entries, "yellow pages" information, and so forth. The impetus for presentation standards comes from advertising and marketing interests.

These applications demand a standard image representation, because manufacturers are reluctant to attempt mass production of the consumer electronics required unless there is a clear standard. A variety of standards are being developed in the U.S. and Europe, collectively known as "videotex" standards [17].

Several different standards efforts are underway:

1. North American Presentation Level Protocol Syntax (NAPLPS). This standard, developed largely in Canada, is being proposed for the United States and Canada [2, 5].
2. CEPT, a European consortium, is developing standards that have evolved from Prestel, the system pioneered in the United Kingdom. Considerable effort is being taken to use character sets that will apply throughout Europe.

An effort is underway to harmonize these two approaches. The principal difference between the two concerns the treatment of graphics: the CEPT effort uses a "mosaic" that approximates images using a special character set, while NAPLPS specifies images geometrically.

Neither of these standards approaches the precision of image definition possible in Interpress. There is no control of font selection, only of character size. Moreover, text and graphics are written on two different "surfaces" that need not be registered very carefully. In short, the low-resolution, low-quality objectives of these standards dominate.

23.3.7 Data storage and transmission standards

While Interpress masters can be stored on floppy disks and magnetic tapes and transmitted via telephone lines, local computer networks and satellite links, the Interpress standard has no appreciable interaction with standards for data storage and transmission. An Interpress master is embedded within these storage and transmission schemes as *data*. To be useful to Interpress, a storage or transmission medium must be able to handle reliably an arbitrarily long sequence of arbitrary 8-bit bytes.

Many Xerox computer-printing products will connect to the Ethernet [20] and accept masters transmitted to the printer using Xerox standard Ethernet protocols. The *Printing Protocol* [28] is used to communicate printing requests to printers and obtain status in return. This protocol is defined in terms of lower-level Ethernet protocols.

23.4 Full Interpress

Interpress version 2.0 is a subset of a complete "Full Interpress" design. While release of the full design awaits more complete testing and evaluation, its principal features are listed here in part so that no one is tempted to "extend Interpress" in various ways. The full design includes, in addition to all features of Interpress 2.0:

- Extensions to allow arbitrary colors defined in several color spaces.
- Extending trajectories to allow arcs and curved segments.
- Extensions to allow precise control over geometric scan-conversion rules when necessary.
- Extending pixel arrays to allow pixel values to have at least 8-bit precision. Photographs can then be represented by the appropriate intensity samples and a printer can choose a suitable halftoning or other rendering technique for printing the image. Pixel arrays may also specify full-color images.
- Extending the imaging model to clip all masks against a geometric clipping boundary.
- Extensions to the base language to allow more kinds of permanent data storage in addition to frames.
- Extensions to the metric master to report more information about the printer's capabilities and its environment.

The full Interpress design retains Interpress 2.0 as a strict subset. That is, an Interpress 2.0 master will be correctly printed by a full Interpress printer.

23.5 Designers

A great many people have contributed to the design of Interpress. This is an appropriate place to acknowledge their efforts.

The Interpress standard was designed by Bob Sproull, Butler Lampson, and John Warnock. Important contributions were made by Bob Ayers, Yogen Dalal, Chuck Geschke, Jim Horning, Dwight McBain, Jerry Mendelson, Lyle Ramshaw, Brian Reid, Mike Townsend, and Doug Wyatt.

As remarked earlier, the design of Interpress is based on the experience of the Press design at the Xerox Palo Alto Research Center. Appreciation is also given to the many people who participated in the design, implementation, understanding, and evolution of Press.

Appendix A

References

[1] American National Standards Institute. *American National Standard Code for Information Interchange.* ANSI X3.4-1977.

United States version of ISO 646. (Section 2.4)

[2] American National Standards Institute. *Draft Proposed American National Standard for Videotex/Teletext Presentation Level Protocol Syntax.* ANSI X3.110-198x.

Current draft of "North American Presentation Level Protocol Syntax," NAPLPS. (Section 23.3.6)

[3] Coonen, J.T. An implementation guide to a proposed standard for floating-point arithmetic. *Computer*, **13**(1):68, January 1980.

A discussion of the proposed IEEE floating-point standard. A list of errata for this article appears in reference [30]. (Section 10.4.2)

[4] Craig, James. *Designing with Type*, Watson-Guptill Publications, New York, 1980.

An introduction to the art of typographic design. Many practical hints; many examples. (Section 10)

[5] Fleming, J. and W. Frezza. NAPLPS: A New Standard for Text and Graphics. *Byte.* Part 1: **8**(2):203, February 1983. Part 2: **8**(3):152, March 1983. Part 3: **8**(4):190, April 1983. Part 4: **8**(5):272, May 1983.

A readable discussion of NAPLPS and its implications. (Section 22.4)

[6] Forsythe, G.E. and Moler, C.B. *Computer Solution of Linear Algebraic Systems*, Prentice-Hall, Englewood Cliffs, N.J., 1967.

Linear algebra useful in understanding and implementing transformations. (Section 13.2.5)

[7] Graphics Standards: Special Issue. *Computing Surveys*, 10(4), December 1978.

This issue contains several articles about the ACM Core graphics standard. (Section 23.3.5)

[8] Hunter, R. and A.H. Robinson. International Digital Facsimile Coding Standards. *Proc. IEEE*, 68(7):854–867, July 1980.

A good survey of facsimile formats. (Section 23.3.3)

[9] Ingalls, D.H.H. Smalltalk Graphics. *Byte*, **6**(8):168, August 1981.

Section 23.2.3

[10] International Standards Organization. *7-Bit Coded Character Set for Information Processing Interchange.* ISO 646–1973 (E).

This document defines a limited character set for information interchange. It is almost compatible with ASCII. The Interpress uses of ISO 646 are restricted to the subset that is compatible with ASCII.

[11] International Standards Organization. *Graphical Kernel System (GKS)—Functional Description.* ISO/DP 7942.

The current draft of the proposed GKS graphics package standard. (Section 23.3.5)

[12] International Typeface Corporation. *The ITC Typeface Collection,* 1980.

A superlative catalog of type faces from one of the best typeface companies. More than 500 different font samples. (Section 9)

[13] Morison, Stanley. *First Principles of Typography,* 2nd edition. Cambridge University Press, 1967. Volume I of the *Cambridge Authors' and Printers' Guides.*

The principles behind the rules of typography. This book is a classic. (Section 10)

[14] Newman, W.M. Display Procedures. *Communications of the ACM,* **14**(10):651, October 1971.

A method for using procedures to represent graphical symbols. (Section 23.2.1)

[15] Newman, W.M. and Sproull, R.F. *Principles of Interactive Computer Graphics.* 2nd edition. McGraw-Hill, 1979.

Introduction to computer graphics, geometric representations, transformations, and raster graphics. (Section 13.2.5, 23.2.3)

[16] Seybold Report. Aesthetics vs. Technology: typography for electronic printers and video displays. **11**(11), February 8, 1982 and **11**(12), February 22, 1982.

An excellent article by Chuck Bigelow on letterform design and typography for raster devices. (Section 9.6.3)

[17] Seybold Report. Viewdata and Teletext. **10**(6), November 24, 1980. Also Telidon and Videotex. **11**(6) November 23, 1981.

Reviews of teletext and videotex, chiefly from the publishing viewpoint. (Section 22.4, 23.3.6)

[18] Seybold Report. Xerox's Star. **10**(16), April 27, 1981.

A review of the Star workstation; see also [19]. (Section 22.1)

[19] Seybold Report on Word Processing. The Xerox Star: A Professional Workstation. 4(5), May 1981.

A review of the Star workstation; see also [18]. (Section 22.1)

[20] Shoch, J.F., Y.K. Dalal, D.D. Redell, and R.C. Crane. The Evolution of the Ethernet Local Computer Network. *Computer,* **15**(8):10, August 1982.

A summary of the Ethernet's operation, protocols, and development. Contains a good bibliography. (Section 22.5, 23.3.7)

[21] Smith, D.C., C. Irby, R. Kimball, B. Verplank, and E. Harslem. Designing the Star User Interface. *Byte,* 7(4):242, April 1982.

(Section 22.1)

[22] Status Report of the Graphics Standards Planning Committee. *Computer Graphics* (quarterly journal of ACM-SIGGRAPH), **13**(3), August 1979.

A detailed description of the ACM Core graphics standard. (Section 23.3.5)

[23] Updike, Daniel Berkeley. *Printing Types: Their History, Forms, and Use—a Study in Survivals,* 2nd edition (in two volumes), Harvard University Press, 1937. Reprinted in paperback by Dover Publications, 1980 (0-486-23928-4 and 0-486-23929-2).

A careful, readable, and exhaustive treatise on the history and use of type fonts. Although almost 50 years old, it is still the best single source for general information about type faces. (Section 9)

[24] Warnock, J. and D.K. Wyatt. A Device Independent Imaging Model for use with Raster Devices. *Computer Graphics*, 16:(3)313–319, July 1982.

A raster graphics package with an imaging model and philosophy similar to those of Interpress. (Section 8.4)

[25] Xerox Corporation. *Character Code Standard.* Xerox System Integration Standard. Stamford, Connecticut; 1982 October; XSIS 058303.

Xerox standard for associating 16-bit numeric codes with character meanings. Also describes a byte-oriented encoding scheme for strings. (Section 9.1.1, 23.3.1)

[26] Xerox Corporation. *Interpress Electronic Printing Standard.* Xerox System Integration Standard. Stamford, Connecticut; 1983 June; XSIS 048306.

This is the reference document for the Interpress standard, version 2.0. It is written with precision and accuracy in mind, and as a result is not always easy to read. This document is the authoritative definition of Interpress. (Section 1.7)

[27] Xerox Corporation. *Interpress 82 Reader's Guide.* Xerox System Integration Guide. Stamford, Connecticut; 1982 May; XSIG 018205.

A narrative companion to an earlier version of Interpress. Much, but not all, of the discussion applies to the current version [26]. (Section 1.7)

[28] Xerox Corporation. *Printing Protocol.* Xerox System Integration Standard. Stamford, Connecticut; 1982 October; XSIS 118210.

Communications protocol for sending a printing request to a printer (Section 18, 23.3.7)

[29] Xerox Corporation. *Rendering Code Standard.* Xerox System Integration Standard. Stamford, Connecticut; 1982 October; XSIS 068303.

Xerox standard, companion to [25], that assigns numeric codes to different graphical renderings of characters. (Section 9.1.1, 23.3.1)

[30] —. A Proposed Standard for Binary Floating-Point Arithmetic. *Computer*, 14(3):51, March 1981.

A draft of the reference document for the proposed IEEE standard. The same issue of *Computer* also contains other articles about the standard. (Section 10.4.2)

Index

Page numbers inside braces { } refer to pages in the reference document, *Interpress Electronic Printing Standard.*