XEROX

Xerox System Integration
Guide

Interpress 82
Reader's Guide

XEROX

Xerox System Integration Guide

Interpress 82 Reader's Guide

Jerry Mendelson

XSIG 018404
April 1984

Xerox Corporation
El Segundo, California 90245

Notice

Second Printing, 1984

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 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.

The Interpress 82 Electronic Printing Standard defines the digital representation of material that is to be transmitted to and printed on an electronic printer. Its primary purpose is to provide an accurate specification of the Interpress standard. In doing so it is necessarily terse and specific. The primary purpose of this Reader's Guide is to provide explanatory material on the details of the Interpress 82 Electronic Printing Standard which were not included in the Standard itself.

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

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

Table of contents

Figures

1 Introduction

This chapter describes the organization and content of the Interpress^{TM} Electronic Printing Standard, Interpress 82 version.

1.1 Using the documentation

Interpress 82 documentation includes the *Interpress 82 Electronic Printing Standard* and this companion manual, the *Interpress 82 Reader's Guide.*

The precise description of the Interpress 82 language is to be found in the formal specification titled *Interpress 82 Electronic Printing Standard.* Such precision requires the elimination of all forms of redundancy of information so that concepts are uniquely defined in one and only one place; in some instances this results in closely coupled concepts being separated from each other in their presentation. The requirement for precision also results in the use of a formalized syntax which may be unfamiliar to the reader, and in general demands a rather formal style which can be difficult to grasp on first reading.

All of these factors point to the need for a companion document that presents a comprehensive overview of the language, as well as expanded narrative explanation. This manual, the *Interpress 82 Reader's Guide,* is intended to serve that purpose. Part I of the Reader's Guide contains an overview of the basic concepts of Interpress, and Part II provides an interpretive reference key that provides expanded descriptions of the more complex concepts presented in the specification; Part II has paragraph numbers which are in one-to-one correspondence with those of the specification, and can readily be used as a reference source for amplification and clarification of the formal document.

Note: The *Interpress 82 Electronic Printing Standard* is the authoritative source for the Interpress language definition. Any conflict of interpretation between the companion documents and the *Interpress 82 Electronic Printing Standard* is to be resolved by reference to the latter document.

2 Basic principles

Interpress 82 is a language that Xerox has created expressly to define the contents of a document to a Xerox raster scan type of printing system. It is a concise, accurate, printer independent language that can be readily processed by most Xerox printing systems of the raster scan type.

2.1 Interpress characteristics

Interpress 82 has the following important characteristics which make it particularly useful:

1. Its structure is simple enough to permit both low end printers driven by microprocessors such as the 8086, and high end printers such as the 9700, to absorb and still maintain their full throughput capability.
2. Its language is rich enough to permit the description of an extremely large range of black and white images.
3. Its language is extensible so that new printing constructs may be added over time.
4. It is supported by most Xerox printers and print generating sources so that documents may be exchanged among most Xerox products.

2.2 Functional concepts

A *document* is represented in the Interpress 82 language by a non-pictorial digital program called an Interpress *master*, which, when executed by the printer's software, directs the precise imaging and printing of each page of the document. Within this master, each document page is independently programmed, so any page of the document may be printed in any order without first having to image, or analyze the images of, the preceding pages.

The function of an Interpress master is to describe precisely the desired appearance of *a page which has been completely composed by some other process.* All font selection decisions, line ending decisions, hyphenation decisions, line justification decisions, graphics positioning decisions, and any other forms of information positioning decisions must have been made by a host processor (referred to as the *creator*) prior to the invoking of the process which creates the Interpress master. The only positioning adjustments the creator may invoke at the printer are minor *readjustments* to the positions of characters on a line so as to present a line of text which has been optimally adjusted to the actual font widths and resolution capabilities of the printer. This gives a sophisticated printer the opportunity to take advantage of its own special knowledge of its fonts and its resolution accuracies for either of two reasons: to adjust out minor differences between the font width information assumed by the creator and the font width information actually used within the printer, or to permit the printer to do a reasonable job of printing a page when it must substitute a totally different font for the one specified by the creator.

Interpress allows the creator to specify a document's ideal appearance in a printer implementation independent fashion. The appearance of the final document is completely governed by the creator of the master; the printer software that interprets the master makes no formatting decisions, but may make device dependent positioning decisions of the type described above. That is, the target printer may not be able to position information to the precision called for, or might not possess exact copies of the fonts specified, but it will provide output images which are as accurate as it can within the constraints of its capabilities. Files containing documents described in the Interpress language may be manipulated by utility programs (external to the printer and prior to their transmission to the printer) to provide capabilities such as merging the pages of one document with those of another, merging onto one page image information from another page image, transforming an image's representation so that it is rotated on the page, transforming two images so that they are merged onto a single image in a "two-up" format, and so on.

2.3 The Xerox encoding system

An Interpress master is encoded in a transmission format that allows it to be communicated to Xerox printing systems accurately and efficiently. A formal description of the *Xerox Standard Encoding System* is included with the specification of the Interpress 82 language; however, the syntax of the encoding system is separate from and different than that of the language proper. *Interpress syntax* refers to the structure of the executable language elements; *encoding syntax* refers to the transmission form of the master.

2.4 Interpress imaging capabilities

Interpress 82 is designed to represent all kinds of images within the conventions of one standard language.

The images to be printed on a document page can be generally classified as *character coded*, *vector graphic*, and *pixel arrays*, defined as follows:

Character Coded -- Text; strings of characters in a chosen series of fonts, typically created by a document processing system to convey verbal information.

Vector Graphic -- Line drawings specified via trajectory coordinates.

Pixel Arrays -- Pictures represented by bit maps, typically created by a raster scanner.

These classifications are descriptive of a representational mechanism, and are not necessarily related to the pictorial characteristics of the printed image. For example, a pixel array could be used to represent a page of text. The Interpress language can describe character coded, vector graphic, and pixel array images, singly or in combination.

2.5 Printer implementation capabilities

Printers are not universally capable of printing every conceivable kind of image that Interpress 82 is capable of describing. For example, some printers can handle vector graphics but do not support pixel arrays. In order for an Interpress user to be able to anticipate the implementational constraints imposed by a particular printer, a method of identifying the implementation level of the printer is provided. (See "Support Levels and Mapping" in Appendix B.)

This does not mean that an Interpress master must be tailored for a specific printer; one of the key design features of the Interpress 82 language is the ability to default across unsupported constructs without unnecessary print interruptions, catastrophic error conditions, or unpredictable side effects.

2.6 Document processing scenario

The generation of an Interpress document can be traced through the following steps:

1. The content of a document is established in some host processing system known as the creator.
2. The creator arranges the content of the document into its final desired appearance, in which all of the formatting decisions, page layout decisions, font selection decisions, graphic incorporation decisions, and so on, have been determined.
3. At this point the Interpress language enters the picture. The creator produces an Interpress description of the finalized document, thereby providing a precise, device independent document master.
4. In order to send the Interpress description of the document to a printing system, the master is encoded into a compact transmission format. Within this encoding, the Interpress master is expressed as a series of binary digital sequences, and packaged into units of transmission called *tokens* along with identification information that allows them to be decoded upon receipt at the printer. (As a practical matter the conversion of the composed document into the Interpress language, and its further conversion from that language into its token-encoded form, takes place in a single step within the host processor.) Note that Interpress is a language which is intended for machine, not human, generation, so no programmer-oriented mnemonic set of literals representing constructs of the Interpress language has been defined within the formal language specification.
5. The encoded Interpress representation of the document is then transmitted to a printing system.
6. Software within the printer receives and decodes the encoded representation of the document. It then proceeds to interpret the Interpress language description of the document and create the page images called for by that language description.

Since this software is interpretive in nature it can, if necessary, leave the Interpress interpretation process at any time to carry on activities which are printer-related. For example, the Interpress description of the document may call for the use of character fonts which it presumes to be resident in the printer. When such fonts are called for, the Interpress interpreting software in the printer leaves the Interpress interpretation process, goes off to the printer's font library to locate the desired fonts, sets them up for use by the Interpress interpreting process, and then returns to that process. Similarly, it can leave the Interpress interpretation process to examine printing instructions, to incorporate non-Interpress files, or to carry out any other printer related activities. Such activities can be invoked by operations within the master itself.

From the above description it should be clear that the creator of the master and the encoder of the results of that creation reside in a host system. The interpretation of the master, and its conversion to the printed document which it describes, occurs in the printer. The printer software activities are not limited to those of interpreting the master. The master may contain operations which invoke the execution of activities outside itself.

2.7 Error reporting

As a document is printed various errors may occur. The user is always informed of an error, for example by printing an explanatory message on a header or trailer page that accompanies the document. There are four types of errors:

1. Appearance Warning. The imager had to make an approximation of the ideal image represented in the Interpress master, but has been able to preserve the content of the image. For example, if a different font has been substituted an appearance warning is generated.
2. Appearance Error. The imager had to make an approximation to the ideal image represented in the Interpress master in such a way that the resulting image will not appear to be correct. For example, if an imager cannot display a raster scanned image represented by a pixel array that is called for in the master, an appearance error is generated.
3. Master Warning. Something is amiss in the specification of the master, but the error is not severe. For example, if an arithmetic overflow occurs in the processing of a master, or the master attempts to make a reference to a non-existent component of a vector, a master warning is generated.
4. Master Error. These errors signal severe problems in interpreting the master. It may be necessary to abandon further effort to print the master, and to simply print a document identification page that describes the error.

The next sections describe the transmission elements of the encoded master, and the executable Interpress language elements that are derived from them. Following that is a description of the programmatic structure and implementation of the Interpress master.

3

Elements of the encoded Interpress master

This chapter describes the standard Xerox encoding system that is used to encode the contents of an Interpress master, and presents the elements of an encoded Interpress master. The Xerox encoding system referred to in this context is not to be confused with the Xerox Character Encoding Standard. The latter defines a system for encoding a sequence of 16-bit codes that uniquely define characters and symbols for imaging. It is used *within* the Xerox encoding system in dealing with the type *string* described below.

3.1 Xerox encoding

When a host processor creates an Interpress master it puts the Interpress language constructs and the imaging data that make up the master into an encoded form that can be rapidly transmitted and efficiently absorbed by a printer. The Xerox printers that support Interpress also support an encoding standard called the Xerox Encoding. This encoding standard is precisely defined in the specification along with the Interpress language, but it should be noted that the encoding conventions are separate from the Interpress language syntax.

3.2 Header information

Alternative encoding systems may be created either by Xerox or by other users of the Interpress language, so every master begins with a header which unambiguously identifies the encoding standard used within the following document. The header for all encoding systems must be expressed in a series of bytes, with each byte containing a character from the ISO 646 7-bit Coded Character Set for Information Interchange. The high order bit of each such byte is a zero. A "space" code acts as the terminator for the header, which therefore cannot contain a "space" code as one of its characters. The header for the standard Xerox encoding is the following ISO 646 character sequence, each character contained within one byte:

Interpress/Xerox/1.0ƀ

(where the "/" character is part of the header, and the ƀ stands for the "space" code which terminates the header.)

This string of 21 bytes must be the first 21 bytes in the document, or a printer expecting the Xerox encoding header will reject the document and report an error.

3.3 Tokens

The standard Xerox header is followed by a series of *tokens.* Each token is a transmission sequence of bytes containing information that the printer's software can decode and build into Interpress language constructs and imaging data needed to print the document. The beginning of the token contains descriptor fields that allow the printer to correctly decode the token. Following the descriptor fields are data bytes containing information used to describe the document to the printer.

3.4 Token formats

Each data byte in a token is an 8-bit binary integer in the decimal range of zero to 255. The most significant byte of the transmitted token arrives at the printer first, so that if its bytes were printed on a page in a left-to-right fashion in the order in which they are received the entity would be in its natural reading order in the English language.

These tokens have five different formats, as illustrated in Figure 3.1. The first descriptor field within the token identifies the token format to the printer. Three of these formats (*Short Op, Long Op*, and *Short Number*) are of fixed length; the remaining two formats (*Short Sequence* and *Long Sequence*) are of variable length, and are used to transmit long sequences of bytes. Both the Short Sequence and Long Sequence tokens contain information encoded in one of several different ways according to the source and nature of the data; therefore, these last two formats contain descriptor fields which define the type of encoding representation used for the data byte sequence which the token contains, and define how many data bytes are in that sequence. Because of the presence of the length field in the variable length tokens, and the fixed length of the other tokens, the exact length of every token in the input stream is known by the printer as soon as it has received no more than four bytes of a token.

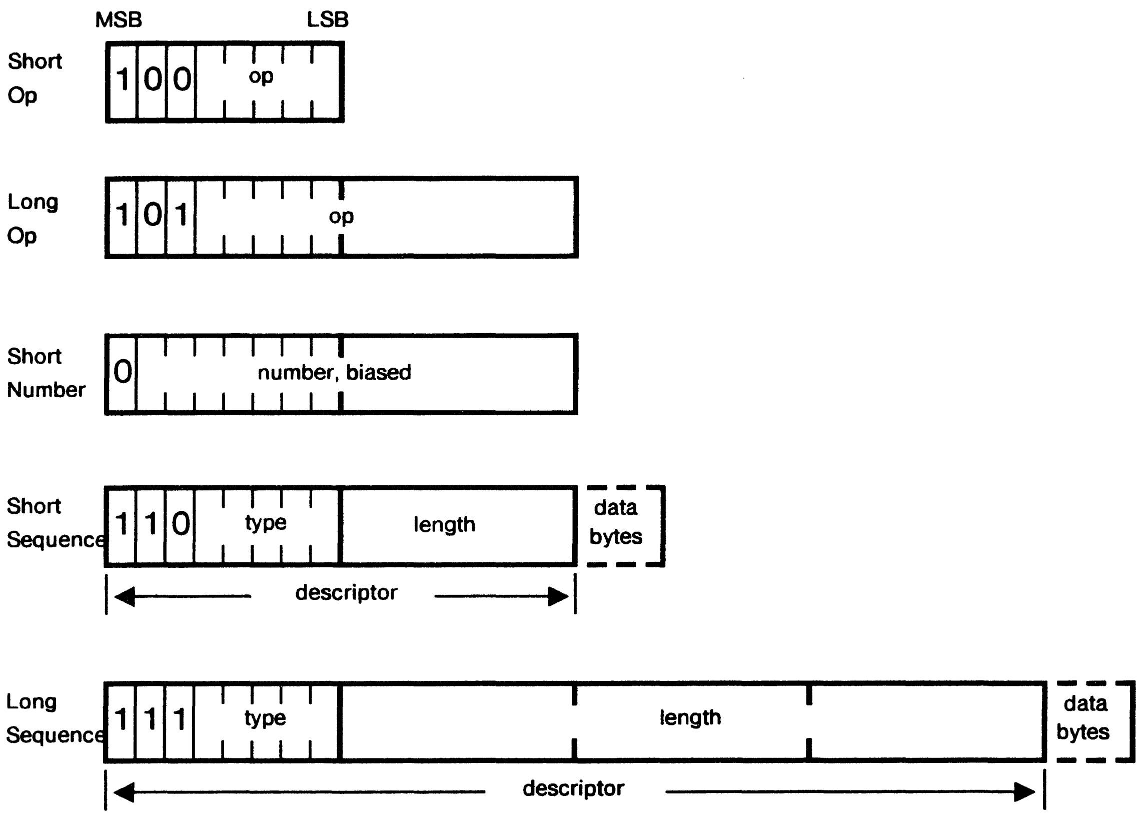


Figure 3.1 Token formats

The following paragraphs describe each of the five token formats in detail.

3.4.1 Short Op and long Op tokens

The Short Op and Long Op tokens are used to transmit Interpress operators and the codes for "BEGIN", "END", "{" , and "}" to the printer.

The Short Op token is a one-byte token containing a fixed 3-bit token identifier, 100, followed by 5 data bits containing any of the 32 most frequently used opcodes, encoded as binary integers in the decimal range zero to 31. The Long Op token is a two-byte token containing a fixed 3-bit token identifier, 101, followed by 13 data bits containing any of the 8192 possible opcodes whose assigned integer codes are in the decimal range of zero to 8191, or any of the codes "BEGIN", "END", "{" , and "}". An operator whose encoding representation is in the decimal range of zero to 31 may be encoded in either of these two token types.

The Interpress 82 Electronic Printing Standard lists the assigned code values for these operators and delimiters.

3.4.2 Short number tokens

The Short Number token is used to transmit an integer ranging in decimal value from -4000 to 28767. It is a two byte token containing a fixed 1-bit token identifier, 0, followed by 15 data bit locations which contain the encoded integer.

An integer in this range is encoded by adding the decimal value 4000 to it, and then converting the result to an unsigned 15-bit binary integer. Thus, the value -4000 is represented by 15 binary zeros, and the value 28767 is represented by 15 binary ones. An intermediate value such as -125 becomes the 15-bit binary equivalent of -125 + 4000 = 3875, which is 111 100 100 011. An integer outside the range -4000 to 28767 is encoded in a Short Sequence or Long Sequence token of type *sequenceInteger*, described below.

3.4.3 Short sequence and long sequence tokens

The Short Sequence and Long Sequence tokens are used to transmit extended sequences of data bytes. The Short Sequence token begins with a 3-bit token identifier, 110, followed by a 5-bit type field which defines the type of encoding representation, and an 8-bit length field, which defines the number of encoded data bytes following the descriptor fields. (The length field does not include the bytes in the descriptors.) The Long Sequence token begins with a 3-bit token identifier, 111, followed by a 5-bit type field, and a 24-bit length field.

Eleven possible types of encoding representations may be used to represent the data transmitted within Short Sequence or Long Sequence tokens. The 5-bit type field contains a numerical value identifying which encoding representation has been used; these values are listed in *The Interpress 82 Electronic Printing Standard*. The content of the data field for each type of representation is described below.

sequenceInteger: Contains an unsigned two's complement binary representation of an integer. This type is used to encode an integer outside the range -4000 to 28767. It may also be used to encode an integer within that range, but provides a less compact form for doing so.

sequenceRational: Contains two signed two's complement binary representations of the integer numerator and denominator of a rational number. The numerator precedes the denominator in the encoding. Each integer is half the length indicated in the token's length field; that is, the data field is twice as long as the number of bytes required to hold the longer of the two integers. Both integers are padded with high order zeros to fill out their allocated space.

sequenceIdentifier: Contains the characters of an Interpress identifier (which is defined in the Interpress language syntax section of this manual), encoded in the same fashion as the characters of the header. Each byte contains a character from the ISO 646 7-bit Coded Character Set for Information Interchange.

sequenceLargeVector: Compactly represents the components of a large vector in the form of a set of two's complement binary integers, with each integer representing a component of the vector. Each integer in the set is the same length and is represented by a series of bytes. The token's first data byte following the length field contains a value indicating the number of bytes in each of the succeeding integers; if the number of bytes is *b*, and the number of such integers is *n*, then the first data byte will contain the value *b*, and the length field will contain the value $n*b + 1$ (where the additional 1 occurs because of the data byte containing the value *b*). This set of integers is automatically converted to an Interpress vector (defined in the Interpress language syntax section of this manual) at the printer during the token decoding process.

It generally is not practical to transmit an extremely large vector in terms of its individual components; the sequenceLargeVector encoding representation forewarns the printer's token decoding process that an extended sequence is being transmitted, and packages the vector into one token. The decoder can then build the vector as it arrives, store it in internal or external memory, and use a pointer technique to represent its location.

sequenceCompressedPixelVector: Contains a raster which has been compressed in specific accordance with the Xerox standard compression algorithm that is defined in Appendix C. (No other algorithms are currently supported.)

The sequenceCompressedPixelVector represents the compressed raster as a set of 32-bit fixed length binary integers, with each integer representing a component of a pixel vector. This token has exactly the same structure as the sequenceLargeVector described above; the value of *b* in this case is 4, representing the four bytes in each 32-bit integer. Its maximum length is 16777215 (2^{24}-1) bytes. The maximum number of 32-bit integers that it can contain is therefore 4194303 (2^{22}-1).

sequencePackedPixelVector: Contains an uncompressed raster, each component of which is represented as a 32-bit fixed length binary integer. Components are packed according to the following specific rules:

- Consecutive pixels along a scan line are stored in consecutive bits within consecutive bytes.
- The end of the scan line is padded with zeros to bring the scan line ending to a byte boundary.
- The end of the last scan line is padded with zeros to bring the end of the complete raster to a 32-bit boundary.

(No other packing algorithm is currently supported.)

This token has exactly the same structure as the sequenceLargeVector; the value of *b* in this case is 4, representing the four bytes in each 32-bit integer. Its maximum length is 16777215 (2^{24}-1) bytes; therefore the maximum number of 32-bit integers it can contain is 4194303 (2^{22}-1).

sequenceContinued: Contains a continuation of the preceding token's data field. A token of the sequenceContinued type may be used regardless of the kind of token preceding it, but it is typically used in conjunction with the sequenceCompressedPixelVector or sequencePackedPixelVector types.

The sequenceContinued type is provided because although the encoding system always supplies the precise length of each token, the total length of a large set of vector components may not always be known at the time that its transmission must begin. This can occur, for example, in scanning devices which do not contain buffers large enough to hold a complete image. Such a device generally uses a double buffer or ring buffering scheme. At the time it begins to transmit the first buffer segment of that image, it knows the length of each buffer that it has prepared for transmission, but not the total length of data in the scanned image. The first such buffer that is filled is transmitted as a sequenceCompressedPixelVector or sequencePackedPixelVector, with its length field properly set to reflect the buffer size. Subsequent buffers that are filled are transmitted using the sequenceContinued type with each length field again set to reflect the buffer size. The last segment of the image may not completely fill the buffer, but its size in bytes will be known; it is also transmitted with a sequenceContinued type, with its length field set to reflect the byte count in the last buffer. (The length field of a sequenceContinued type can contain a value of zero.)

sequenceString: Compresses 16-bit components of a vector into a string of 8-bit values, in accordance with a standard compression algorithm. An alternative compression algorithm documented in Appendix C can be used for any set of 16-bit components of a vector; however, the sequenceString format typically is used to represent characters that are defined by 16-bit character codes, such as those defined in the Xerox Character Code Standard. (See the Xerox Standards Document titled *Xerox Character Code Standard.*)

The contents of a sequenceString are 16-bit (two-byte) quantities that are represented in a compressed format. The compressed format encodes two-byte quantities into one-byte quantities by establishing a constant value for the high order byte. This constant value is then concatenated to all succeeding bytes to form two-byte quantities. The constant value high order byte is changed to a new value by an escape code sequence. The escape code is a byte with the decimal value 255. When the escape code is detected the following byte is established as the new value for the constant high order byte. The rules for this process are the following: if the token's leading data byte contains all ones (decimal value 255), the next byte designates the high order byte of all the two-byte vector components; each of the following series of bytes is assumed to be the low order byte of each two-byte component. If the leading byte is not 255, the upper byte of each component is assumed to contain all zeros. Once the value of the upper byte has been defined it remains in effect as the upper byte of all succeeding two-byte components until it is changed by the appearance of a byte containing the decimal value 255. The upper byte may not be assigned the value 255.

Xerox supports a standard 16-bit encoding convention for its Xerox Character Code Standard, which comprises up to 255 different character code subsets, each containing 255 characters. (The 256th character code in each subset, having code value 255, is reserved for the special use described above, and may never be used as a code set designator nor as a character code within a code set.) The high order byte of this 16-bit character code designates a character code subset, and the low order byte designates the character code within that subset.

Xerox also supports a standard set of characters called the Xerox Standard Character Set. The Xerox Standard Character Set is broken up into as many as 255 different subsets, each containing up to 255 characters. Each member of the Xerox Standard Character Set has an associated member of the Xerox Standard Character Code by means of which its imaging may be invoked. The relationship between the members of the Xerox Standard Character Set and the Xerox Standard Character Code is presented in the Xerox Standard titled *Xerox Character Code Standard.* Reference to this document will show that character subset zero includes the ISO 646 set. Therefore, a conventional ISO 646 string is represented in a sequenceString encoding by a string of bytes corresponding directly to the ISO 646 byte encoding system. The first byte of such a string will not contain the value "255", hence all the bytes of the ISO 646 string will have an implied "0" byte concatenated to them in the creation of their 16-bit character codes.

sequenceInsertFile: Contains a sequence of 8-bit character codes representing the name of a file in an installation dependent character set. For Xerox printers that is the *Xerox Standard Character Set* .

When a sequenceInsertFile token is received by the printer, the software executing the Interpress master in the printer essentially leaves the Interpress domain. It obtains the designated file and determines the nature of its content. It can then take any number of different actions depending on the content of the file. The file content may be anything the installation chooses to support, subject only to one constraint: if the file causes the state of the machine to change or causes an image to be created in any manner whatsoever, that image must have been definable by an Interpress master, even though it might not be so defined in the file.

Such a file can contain a number of different types of information, including, for example:

1. A fragment of an Interpress master. The fragment is inserted in the master in place of the sequenceInsertFile token and executed as though it were part of the master itself.
2. A complete Interpress master. The printer software can strip off the header and the master's preamble (see 3.5), and execute the rest of the filed master in-line with the master currently being transmitted. Alternatively, it might execute the preamble and create a special PageFrame for use in the execution of the master contained in this file. At the completion of the execution of this file it would have to restore the old PageFrame for the continuation of the execution of the original master.
3. A non-Interpress representation of a page image. Generally, each printer does not transform an Interpress master to a raster image format directly; it usually generates some intermediate native language to drive its image generating system. Once a master has been interpreted and this native language format has been generated, the result may be stored in a file for later re-use. The printer software can recognize such a file and place its content directly into the native language output stream that it is generating from the Interpress master.

A standard form is an excellent example of the utility of the sequenceInsertFile token. Within a given complex of printers there may be many different printers, each with its own native language. The same standard form might be contained in a file in each printer, in that printer's native language. Each such file could have the same file name. Now a master containing a sequenceInsertFile token specifying that file name would create the same image at each printer even though the contents of every file would be different.

This function is one of the most powerful tools available to the creator of an Interpress master. In essence, the sequenceInsertFile operation provides a means for escaping momentarily from the Interpress language. Its use is analogous to that of escaping from a higher level programming language to an assembly language level.

sequenceComment: Contains an arbitrary sequence of bytes which are ignored by the printer when this token is processed.

3.5 Organization of the encoded master

Within the Xerox encoding format, the encoded master is organized into groups of tokens containing operators and operands that describe the individual pages of the document. Constants and procedures common to all pages may be included in a *preamble*; except for this common information, each page is independently encoded and contains both the procedures and the data constants and imaging literals that the printer needs to build that page. Note that the master does not provide a central program that loops through a new set of data variables for each successive page; rather, each page is a separate program wherein page imaging data is included as a fixed collection of literals. (This allows the user to select part of a document or a series of nonsequential pages for printing, without requiring the printer to process preceding pages that are not to be printed.)

The tokens of the encoded master are grouped into pages through the use of the special delimiter codes represented in this manual by the brace characters { and } ; tokens containing BEGIN and END codes delimit the master itself. (*The Interpress 82 Electronic Printing Standard* lists the code value of each of these delimiters; they are typically encoded using a Long Op token format.) Thus, the structure of a master is as follows:

header BEGIN{preamble}{page 1}{page 2}. . .{page n}END

The primary function of the preamble is to select and store the fonts; it also establishes parameters, procedures, and working storage used throughout the printing of the document. The preamble is within the first set of brackets following begin. If no preamble information is supplied, the preamble's position must be acknowledged with a pair of empty braces, as follows:

header BEGIN{}{page 1}{page 2}. . .{page n}END

3.5.1 Order of notation

An opcode normally must *follow* its associated operand or operands in the encoded master. This is referred to as "postfix notation", and allows most printers to absorb and execute the encoded master efficiently.

Within the encoding system a very small number of specific opcodes *precede* a complex operand called a *body*, which in encoded form consists of brace codes delimiting a group of opcodes and their associated operands. The entire group is treated as a single operand of the body operator. It is encoded as follows:

Body operator { operand(s) operator . . . operand(s) operator }

The preamble and pages of a master are special instances of this structure, wherein the body operator is implied rather than explicit; executing a *page body* causes the page to be imaged and printed.

(It is important to note that this precedence characteristic of a body operator is an *encoding* convention and does not reflect the syntactic rules of the Interpress language structure itself, as described in the following chapter. In fact, Interpress language syntax requires that the body operator immediately follow its associated operand, and that no operands or operators occur between a body and its associated body operator. It is this syntax restriction which permits the body operator to precede its body operand in the encoding system.)

4

Interpress language components

In review, Interpress 82 is a language for representing a previously composed document, wherein Interpress operators (procedures) and operands (data constants such as page location parameters, and imaging literals such as text characters) form a programmatic master of the document. The master is transmitted in encoded form to a printer, which decodes, interprets, and executes it, thus producing a printed copy of the composed document.

This section describes the executable Interpress language components that are derived from a transmitted master by the printer's software.

4.1 Transmitted and created language components

Transmitted tokens are decoded by the printer and the data they contain is built into an executable sequence of Interpress operators and their associated operands. There is not always a direct one-to-one correspondence between an encoded token and an Interpress language component; some components are built at the printer as a result of the execution of directly encoded components.

A component of the Interpress language is called a *transmission type* if it can be encoded in a token and transmitted to the printer, or a *creation type* if it is built at the printer as a result of the execution of transmission-type components. For example, there is an Interpress language component called a *trajectory*, but an Interpress trajectory cannot be directly represented in the encoded data bytes of a single transmission token. However, a number of separate tokens can be transmitted which contain encoded operators and operands that cause a trajectory to be built at the printer. Thus, a trajectory is a *creation*-type language component which could not have been directly encoded and transmitted, while those which built the trajectory were *transmission* types and were directly transmitted. Language components of the transmission and creation types are collectively referred to as *execution type* components.

4.2 Data types

An execution component of Interpress 82 carries an internal tag identifying it as one of twelve separate data types, according to its form and function. Transmission language components may be any of four data types; creation language components, formed within the printer, may be any of nine different types. (The Operator data type is used to identify both primitive operators, which are transmitted, and composed operators, which are created at the printer.)

4.2.1 Transmission data types

The following are the different types of transmission components, each of which may be encoded and transmitted in a token.

Operator: A transmission-type operator is called a *primitive* operator; it is a basic procedure used to build an output image or control the intermediate data used in the process of building it. Normally an operator is executed upon receipt; however, it may be contained within a complex operand called a *body* (see below), in which case it might not be immediately executed. The individual operators are described in the specification, along with the number, sequence, and type of operands they require; note that when a particular operator requires a *number* as an operand, that operand may be either type Number or type Integer as described below. (An operator is encoded as a Short Op or Long Op token.)

Some operators are created at the printer as the result of the execution of other operators; see 4.2.2.

Integer: A 24-bit unsigned mathematical integer that must lie in the range zero to 2^{24}-1. Whereas a token-encoded integer may be represented by the smallest number of bytes possible, it is expanded to a 24-bit form after it is received. (It is encoded as a Short Number token, or as a Short or Long sequenceInteger token. Note that an Integer as an Interpress language type is different from an integer in the encoding system. In the encoding system an integer may be negative, and, if it occurs in the definition of a rational, may have a magnitude greater than 2^{24}-1. Any encoded integer outside the range 0 to 2^{24}-1 is converted to a value of type *Number* during the input decoding operation.)

Number: A Number is an element of a certain subset of mathematical rationals. This subset includes every rational which can be represented in the form $ix2^e$, where i is an integer in the range -(2^{24}-1) to (2^{24}-1), and e is an integer in the range -100 to 100. Numbers may be transmitted in several ways. If they are integers there are two transmission mechanisms. If they are not integers they are transmitted in terms of *rationals.* A *rational* is a pair of integers named *numerator* and *denominator.* Integers used to define rationals are mathematical integers, not necessarily Interpress integers. That is, they are not restricted to the range 0 to 2^{24}-1, but may be signed, and of greater magnitude range. Most printer implementations will deal with integer values in the range -2^{32} to 2^{32}-1. The number represented is *numerator* divided by *denominator.* In the Interpress language rationals or integers may be used wherever the formal operand type *number* is called for. (Numbers are encoded as a Short or Long sequenceRational token, or as a Short Number token.)

Identifier: A sequence of characters used as a constituent of a hierarchical name. Each character is encoded within one byte in the ISO 646 7-bit Coded Character Set for Information

Interchange, the high order bit of the byte being set to 0. An identifier may only include lower-case letters, digits, and the "minus" character (-), and must begin with a letter. Upper-case letters may be included in a token-encoded identifier, but they are mapped to their lower case equivalents when they are received. The Interpress language makes no provision for decomposing an identifier into its consituent characters; it is treated as an atomic entity. (It is encoded as a Short or Long sequenceIdentifier token.)

4.2.2 Creation data types

The following list summarizes the different types of creation language components, which may not be transmitted as single corresponding tokens in the encoding system, but may be built at the printer using the transmission components described above.

Body: A complex operand that contains a series of operators and their associated operands. When a body is received at the printer it is immediately acted upon by its own associated *body operator*, which must be one of the following: MAKESIMPLECO, DOSAVESIMPLEBODY, IF, IFELSE, IFCOPY, CORRECT. (In the encoding system, a body is delimited by beginning and ending brace codes and the body operator is encoded preceding the body. In the internal language syntax, however, the body operator is required to immediately follow the body. No other operator nor operand may be interposed between the body and its associated body operator. It is this constraint which permits the body operator to precede its body within the encoding system.)

Operator: A creation-type operator is called a *composed* operator; it is a procedure composed of a body and a set of constants and variables called a *frame*. (Refer to "composed operators and the active frame" for additional details.) It is built as the result of the execution of a MAKESIMPLECO operator.

Vector: A collection of ordered execution language components, built as the result of the execution of a MAKEVEC or MAKEVECLU operator. An Interpress operation can access a vector element by indexing its position (the first element, the second element, etc.). The elements of a Vector are not required to be of the same data type; they may be of any type, including vectors. (Exception: a body must already have been acted upon by a body operator before a MAKEVEC or MAKEVECLU is received; therefore composed operators, which contain bodies, can be elements of a vector, but the bodies themselves cannot.) Special tokens exist for transmitting the components of large vectors; refer to "Building a Vector on the Stack" for additional details.

Mark: A delimiter which limits access to a common collection of operands called the *stack*. A mark cannot be directly transmitted, but is built as the result of the execution of a MARK operator. A MARK operator is an operator that can be transmitted. (Refer to "The Stack" for additional details.)

Transformation: A special vector of six numbers (of the Number type as defined above), which when combined with three fixed numbers forms a transformation matrix which can be conceptually represented as:

$$\begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

The sequence of presentation of the six numbers is in the order a, b, c, d, e, f. (Refer to Chapter 8, "Transformations", and D for additional details.)

Trajectory: Geometric information describing an ordered sequence of connected lines or curves called *segments.* Segments have a start point and an end point. The end point of one segment in a trajectory coincides with the start point of the next segment in that trajectory. A closed trajectory closes upon itself, i.e., the end point of the last segment in the trajectory coincides with the start point of the first segment in the trajectory.

Outline: Describes a collection of trajectories. Each trajectory of the outline is either closed or implicitly closed by a straight line segment linking the end point of its last segment with the start point of its first segment. It is built as the result of the execution of a MAKEOUTLINE operator.

PixelArray: Contains a raster image plus image characterizing information. It is built as the result of the execution of a MAKEPIXELARRAY operator.

Color: Contains a color description, either of a constant color or a sampled black-and-white color. Interpress 82 supports only the constant color, *gray,* which may range from white to black, or a *sampled black-and-white* color which contains a pixel array of ones and zeros. In the latter case the ones represent "black", and the zeros may represent either "white" or "transparent". Sampled black-and-white color may be used to represent generalized regular patterns made up of vertical, horizontal, or diagonal bars, or dots of various sizes and shapes, etc.

5 Execution state

A generalized machine model is described in the specification to conceptually define the internal logic followed by a printer's software in executing the transmitted master. In addition to the operands, operators, and bodies previously described some key concepts of this model are the *stack*, the *frame*, the *imager variables*, and the *image*. In combination with the procedural context, these define the state in which execution is taking place, and facilitate describing the functional characteristics of the individual Interpress language components.

5.1 The stack

The *stack* is an ordered collection of operands wherein the last one added to the collection is the first one removed. As each encoded operand is received and decoded by the printer, it is pushed on top of the stack. When an operator appears in the input sequence, its operands are popped off the stack and processed according to the semantics of the operator; some operators push resulting values back onto the stack. (Each operator uses a particular number of operands of specific types, which must have been pushed onto the stack in a specific order; likewise, any results returned by an operator are of a predefined number, type, and order.)

Operands which have been popped from the stack are discarded after execution. To preserve them in the stack, they must be copied before they are executed; special stack-manipulating operators are available to do this.

The process described above is illustrated by the following example. Consider the sequence:

```
a b  ADD  c d  SUB  e f  SUB  DIV  MULT
```

wherein the lower-case letters (a, b, etc.) represent operands, and upper-case mnemonics (ADD, SUB, etc.) represent operators. (In this example, each operator uses two operands and returns one value to the stack.)

With this sequence the successive states of the stack would be as follows:

a	After receipt of a
b	After receipt of b
a	
a+b	After receipt and execution of ADD
c	After receipt of c
a+b	
d	After receipt of d
c	
a+b	
c-d	After receipt and execution of SUB
a+b	
e	After receipt of e
c-d	
a+b	
f	After receipt of f
e	
c-d	
a+b	
e-f	After receipt and execution of SUB
c-d	
a+b	
(c-d)/(e-f)	After receipt and execution of DIV
a+b	
(a+b)(c-d)/(e-f)	After receipt and execution of MULT

5.1.1 Building a vector on the stack

A vector is created by a MAKEVEC or MAKEVECLU operator. Prior to the execution of either of these two operators the individual components of the vector are pushed onto the stack; when all of the components of the vector-to-be have been assembled in their proper order on the stack, an integer which designates the total number of such components (MAKEVEC), or a pair of integers designating the lower and upper index value of the vector (MAKEVECLU), is also pushed on.

When the MAKEVEC or MAKEVECLU operator is executed, it uses this sequence of stacked operands to create a single vector operand on the stack that can be pushed or popped in one operation, rather than requiring a separate operation for each component as before. The created vector is structured to permit the retrieval of any of its components by designating the desired component's index number. The only difference between MAKEVEC and MAKEVECLU is that MAKEVEC uses zero as the lowest index number, whereas MAKEVECLU assigns a range of index numbers that starts with the value *l* and ends with the value *u*.

It is not practical to implement a stack of great size in most printer implementations. For that reason it is not practical to make extremely large vectors by pushing an enormous number of components onto the stack and executing a MAKEVEC or MAKEVECLU operator. Special techniques are provided within the encoding system to avoid this problem for certain large vector structures. (See "Token formats.")

5.1.2 Complex operands

The following list of operand types have structures which can be quite complex:

Vector
Transformation
Trajectory
Outline
Body
Operator (composed)
PixelArray

Each of these operands can be quite large, and can consist of many component parts. For example, a vector can have a large number of elements, and each element can be another vector having a large number of elements, and so on. A pixel array can contain millions of bits. An outline can contain many trajectories and each trajectory can contain many line segments. Any of these operands may be pushed onto the stack, or popped off of it, in precisely the same manner as a simple operand such as an integer. (In any printer's particular implementation of the Interpress language the operand might not actually be physically moved to the stack, but only placed there through the use of a pointer to the operand in memory. From the conceptual viewpoint of Interpress, however, the operand is on the stack whether it is physically placed there or only represented by a pointer.) Any of these complex operands is a single element on the stack; it is sent to or removed from the stack with a single push or pop operation. When it is on the stack its components cannot be dealt with by the master on an individual basis.

(A distinction should be made between the Interpress master which describes a document, and the printer software which interprets this description and creates the printed image. There are many processes available within the interpreter which are not available within the master. The interpreter is free to do anything it wishes to decompose the "atomic" entities of the master in order to interpret their meaning, but the master cannot perform the equivalent operations; for example, the printer could examine the individual characters of an Identifier if it chose to do so, but the master cannot.)

5.1.3 Storing stack elements

Elements on the stack can be stored in and retrieved from a special vector called a *frame*, using the FSET and FGET operators. This is of special significance when the element stored is a composed operator, in that it allows a procedure to be executed later using a DO type of operator rather than being immediately executed upon receipt at the printer.

Stack elements can also be used to set values in another special vector containing *imaging variables* that have housekeeping functions relating to the page image currently being built.

5.2 The frame

The *frame* is a dynamic working storage area with the structure of a vector that is created by the preamble. The FGET and FSET operators may be used in the master's preamble and in any of the page bodies to set, retrieve to the stack, or change the values in the frame; an individual component of the frame is specified by an index number.

When the master begins execution, the frame is a vector of fixed size (see *topFrameSize* in the specification), in which all the component values are set to zero. The preamble can store data elements from the stack into the frame as new vector component values, to be extracted later (in the following page bodies) by using the appropriate index number as the operand of an FSET operator; typically some of these frame components are composed operators (see below). Subsequent pages can retrieve these data elements from their frame by means of an FGET operator.

When each successive page body begins execution, its frame initially is the same as the original frame built by the preamble. (This frame is called the *page frame*). This allows commonly used constants and procedures to be built in the preamble and made available to each successive page during execution. Operators in the body of each page can change or add to the copy of the page frame during the course of execution of the page body, but cannot make these components available to any succeeding page. (In other words, every page is executed in a DOSAVEALL fashion (see below), so nothing can be passed from one page to the next.)

In the process of executing the page, modifications to the page frame may be made; the frame as it exists at the current stage of execution is referred to as the *current frame.*

5.2.1 Composed operators and the active frame

When the printer receives a MAKESIMPLECO body operator and its body (consisting of a sequence of operators and their operands), it builds a composed operator by creating a copy of the current frame and linking it to the body. Typically the composed operator, including its body and frame, is then stored in the current frame for later retrieval and execution using a DO form of operator.

A copy of the linked frame becomes the composed operator's initial frame: when the composed operator is retrieved and executed, this initial frame becomes the *active frame*, and the frame that was active up to the point where the DO was invoked becomes inactive until execution of the DO is completed, at which time it again becomes active. Each time the composed operator is invoked, its active frame is a fresh copy of the frame that was current when it was built; at the end of each execution, its frame is discarded.

A composed operator may invoke other composed operators, which may in turn invoke others. Each has its own initial frame, which becomes the active frame during its execution. The FGET and FSET operators affect only the active frame; inactive frames may not be accessed or altered.

From this description it may be seen that the initial frame of a composed operator may provide values known to the creator of the composed operator at the time the composed operator was built. Alternatively, the creator can cause the composed operator to be contained within another composed operator so that it is provided with an initial frame which contains values known within that composed operator at the time the second composed operator is created. The composed operator can obtain parameters and operands from its initial frame, and can store and retrieve local variables in it.

Any invocation of a composed operator must be from the page body or from another composed operator. Since composed operators may be nested to any number of levels there may be an arbitrary number of composed operators in the midst of execution at a given point in time. Since composed operators may be recursively called there may be an arbitrary number of instances of the same composed operator in the midst of execution at a given point in time.

The combination of the current frame of a composed operator and its return link back to its calling DO operator is called its *context.* Each instance of recursively executing composed operators has its own context; the page body in execution has a context which includes the current state of the page frame.

5.3 Imager variables

Imager variables are values which are used to define and control various facets of image generation, such as current position on the page, the font currently in control of the character printing operations, and the color currently being used. They are contained in an internal vector-structured table called the *Imager State*, and can be accessed or modified using the ISET and IGET operators (as well as other "convenience operators" with more specialized functions). Each imager variable has a specific location in the vector, and is located by specifying its appropriate index number. The imager variables and their positions in the Imager State vector are shown in *The Interpress 82 Electronic Printing Standard.*

When the Imager State vector is modified during the execution of a composed operator, the extent to which the changes are retained once the composed operator has finished execution depends on the form of DO used to invoke the procedure. The imager variables are divided into *persistent* and *non-persistent* variables. As described later in this document there are three different forms of a DO operator called DO, DOSAVE, and DOSAVEALL which are used to control the execution of a composed operator. The general function of these operators is not pertinent at this point. What is pertinent about these operators are the terminal states in which their executions leave the persistent and non-persistent imager variables. If DOSAVEALL is used to invoke the composed operator, no changes to the Imager State vector are retained at the end of the composed operator's execution, i.e., all imager variables revert to their previous values. If DOSAVE is used, changes to persistent imager variables are retained, but non-persistent variables are restored to their previous value. If DO is used, all changes persist. This is summarized in the following table:

Operator	Persistent Variables	Non-persistent Variables
DO	Non-Restore	Non-Restore
DOSAVE	Non-Restore	Restore
DOSAVEALL	Restore	Restore

Each imager variable is assigned a standard initial value when the interpretation of the master commences. The preamble (as well as each page body) is executed by an implicit DOSAVEALL; therefore after the execution of the preamble the imager variables are restored to their initialized values, preventing the preamble from directly passing any modified imager state values on to the page bodies. As a further constraint the preamble is not permitted to leave any values on the stack. However, the preamble can store a composed operator in the page frame which when executed will modify the imager variables to the desired values. If each page body begins with the execution of this composed operator, each page body will reset the variables of the Imager State to the values desired by the creator of the master.

5.4 Alterable and unalterable machine states

The *alterable* state of the "machine" which executes Interpress consists of the *stack*, the *contexts* of all of the composed operators currently in the midst of execution, and the current values of the *imager variables*. The *scope* of a composed operator, i.e. the extent of things with which it can deal, is limited to the state of the machine at the time the composed operator is in execution; its scope with respect to the stack may be limited by its caller through the caller's use of the MARK operator.

Executing an operator, either primitive or composed, causes changes in the state of the machine, or in its output (the image being created), or both. These changes may (and generally do) depend on the current state, but they may not depend on the state of the output. That is, the state of the output image can in no way affect the future state of the machine. Output is write only. Thus, the meaning of an operator can be completely defined by two *transition functions*, namely:

a *state transition* function, which defines how a new state of the machine will be created from its present state as a result of executing the operator.

an *output transition* function, which defines how the present state of the output of the machine will be modified as a result of executing the operator.

There exists an *unalterable state* associated with the "machine" which executes Interpress, namely the environment in which it is embedded. In a sense this environment is outside of Interpress, but it does impinge on the execution of the Interpress master; it is from this environment, for example, that the "machine" which executes the Interpress master obtains the fonts called for by the master. The Interpress language itself does not deal with this environment, but its utility depends on the support of processes and usage of data from this environment, such as font look-up.

6 The imaging model

Interpress builds the image of a page by sequentially laying down the components of the image defined by the master. In doing so it obeys a set of rules that define the *imaging model* to which Interpress adheres. The imaging model involves three objects called the *page image*, the *mask*, and the *color*. It is essential that the reader understand this model in order to understand the workings of the imaging operators.

The page image is a two-dimensional image of picture elements (pixels). A pixel is a small region of the output image whose color can be controlled by the printing device independently of all other regions. The page image is accumulated sequentially through the execution of a series of primitive imaging operators. The current state of the page image should be viewed as a receiving medium upon which colors have been deposited to build a portion of the final image, and to which subsequent colors will be applied to complete that image. In Interpress 82 a pixel can only contain a single color, *gray*, which can range through all shades of gray from white to black.

The mask is a stencil which is laid down on the page image. It is first located and oriented so that the shape of the object defined by the cut out portion of the stencil is positioned precisely where the master desires a copy of that shape to be imaged.

The color is an ink which is painted through the cut out portion of the stencil in order to add the image defined by the stencil to the page image.

An incremental step in the building of the page image thus consists of establishing a color for the ink, defining a mask to be used, locating and orienting it at the proper position on the page image, and painting the color through the mask.

Although these are fairly straightforward concepts there are some hidden subtleties in their functional structures which require elaboration. The mask may be defined in either of two ways. One way is in terms of a geometric definition of the outline of the hole in the stencil which represents the mask. This works well when the shape of the hole may be geometrically represented in a simple fashion. Another way is to represent the shape of the hole in the stencil by a pixel array. Such a pixel array could be created to represent the shape of a letter in a particular font style. This form of character representation is generally easier to deal with than a form defined by a geometric characterization of the character's outline. Alternatively it could be generated from the raster scan conversion of a computer generated graphic, or from the raster scanning and half-toning of a continuous tone photograph. Think of a pixel array *in this context* as an area-covering array of binary 1's and 0's where a 1 defines a square hole (of a dimension corresponding to the size of a pixel) through which ink may be deposited, and a 0 represents an area of the same size in which no ink may be deposited. A series of adjacent 1's represents a contiguous area in which ink may be deposited. Think of a pixel as a square area whose sides do not have an associated physical unit of measurement. In an ultimate usage this area will be scaled to a desired size (which might be 3 feet on a side or .001 inches on a side). The actual scaling applied to pixels in most printers is such that the size of a side of the square area represented by each pixel in the array is exceedingly small, generally on the order of 1/200th to 1/1000th of an inch, small enough so that the resolution of the image created by depositing ink through such an array is sufficient for the desired print quality.

The color of the ink which can be deposited through the mask is limited to two varieties in Interpress 82. The first variety is called *gray,* and ranges from white to black. (Note: The term "gray" as used here is a true gray. That is, a value of gray ranging from white to black may be deposited *within* a *single pixel.* Many printers may only permit a *single* pixel to contain a value of black or white. In such printers a *single* pixel cannot contain a value of gray. In this case the printer simulates a gray value by using a regular pattern containing *many pixels* each of which can only contain black or white. In this latter case the human eye integrates the pattern of multiple pixels to obtain what appears to be gray. It is not this latter case we are speaking of when we say that a pixel can contain a value of gray.) The second variety is called *sampled black-and-white.*

Interpress 82 is often employed with the printing of images on monochromatic printers, generally employing black inks. In such printers a pixel can only contain a single value, black or white. (For purposes of the following description we will assume a black ink printer, although it could be any single color.) In such printers either of the two varieties of inks is made by defining its composition through the use of pixel arrays *in a manner which is distinctly different from the use of pixel arrays in defining masks.* Either of these inks is made by creating an area-covering pattern of binary 1's and 0's. In the case of gray ink the structure of the area covering pattern is implicit. That is, a shade of gray is defined by a fraction, f, whose value represents the fraction of incident light that is to be absorbed. Thus, $f=0$ is white, and $f=1$ is black. From this fraction the printer creates a regular pattern of 1's and 0's which will generate a black ink pattern that the human eye will integrate to the desired shade of gray. Note that the printer does not actually contain gray ink, it merely generates a black and white pattern which simulates gray to the human eye. However, the ink is treated as a solid colored ink in the sense that it completely over-writes any image previously placed on the page in any area which it overlaps. In the case of sampled black-and-white the creator of the master explicitly designates the structure of the ink by creating a pixel array whose 1's and 0's represent an arbitrary pattern of black and white.

There are two ways in which such an area-covering ink pattern can be defined. In the first method, as described above, the printer creates the area-covering ink pattern for a gray ink using a creator supplied parameter which defines the desired gray level. The second way is to create an ink "tile", copies of which are automatically abutted to each other by the printer to build up the coverage of a desired area. The creation of the ink tile also specifies the location of its lower left-hand corner on the page. The abutting process which causes copies of the tile to cover the page proceeds from that initial location. Each 1 in the tile pattern represents a black area the size of one pixel. Either of two alternative meanings may be assigned to the 0's in the tile. In the first of these the 0's are considered to contain white ink, in the second they are considered to contain transparent ink. An ink defined by such a tile may be deposited on the existing page image through either of the two types of masks defined above. In the "0 is white" case the areas covered by the 0's in the ink in positions corresponding to holes in the mask will "white" out any image previously deposited in the page image. In the "0 is transparent" case, any image previously deposited in the page image will remain visible in the areas covered by the 0's in the ink in positions corresponding to holes in the mask. It may be seen from this description that a sampled black-and-white ink with the "0 is white" property can be created which is identical to a constant color gray ink. The same ink with the "0 is transparent" property will behave quite differently from the gray ink, however, because in this case it will not obliterate any underlying image in areas corresponding to 0's in the ink and holes in the mask.

Usually the tile is designed so that when copies are abutted to each other on any side the pattern repeats itself in a continuous fashion throughout the area to be covered, but that condition is not enforced by Interpress. If the continuous pattern structure is inherent in the design of the tile the abutting process ensures that all objects printed on the page with that ink will have a common "phasing" with respect to that ink. That is, the ink pattern on nearby objects will visually flow from one object to another without discontinuities in the pattern. For example, if the ink is of a "stripe" variety the stripes which cover one object will be aligned with the stripes which cover nearby objects, and not offset so as to create visual discontinuities between nearby objects.

In the case of gray ink the pattern of 1's and 0's in the area-covering pixel array of the standard tile is completely homogeneous in all directions. In the case of sampled-black-and-white ink it need not be. In this case the pattern of 1's and 0's could be that of horizontal stripes, vertical stripes, diagonal stripes at various angles, dots of various sizes and shapes, and so on. Again, for esthetic reasons, it is generally the case that the tile is designed so that a series of abutting tiles would present a uniform pattern, e.g. the stripes of each tile would flow contiguously into the stripes of its neighbors.

Finally, a sampled-black-and-white ink could be made from an arbitrarily sized rectangular pixel array with an arbitrary distribution of 1's and 0's. Such an ink could be *generated from the raster scan conversion of a computer generated graphic, or from the raster scanning and half-toning of a continuous tone photograph.* The preceding point is italicized to emphasize the fact that this method of defining an ink is identical to a previous definition of a method for defining a mask. A pixel array can be used to represent either of these elements.

Visualize that you have an ink well containing any of these type of inks. Then visualize that you have a paint brush that you can dip into this ink well. Now visualize that you are going to "paint" this ink through a stencil with an 8-1/2 by 11 inch hole in it onto an 8-1/2 by 11 inch piece of paper. You would have an ink well and a paint brush which you would otherwise only be able to see in a Walt Disney cartoon. You would be able to cover the paper with any shade of gray from black to white, or with a regular pattern of stripes of various thicknesses and various angles, or with polka dots, or with a scanned image copy of the Mona Lisa, all with a single dipping of the brush.

Now visualize that instead of having a stencil with an 8-1/2 by 11 inch hole in it you have a stencil of either of the varieties defined above. You locate and orient this stencil on the page image, and then you paint the ink in your inkwell through the holes in this stencil. Again, note that in this scenario a pixel array may be used in both of the contexts described above. One pixel array may be used to define the mask. A second, completely independent, pixel array may be used to define the ink. Normally one thinks of the mask as having some regular or recognized shape, e.g. a square, a rectangle, a triangle, or a letter or a number. Normally one thinks of the ink as white, black, or a shade of gray. The result of depositing the ink through the mask is then quite easily visualized. You get an image having the shape of the mask, and the shade of the ink.

Conceptually, the imaging model permits completely flexible combinations of masks and inks. The mask can be a rectangle and the ink striped, in which case you can get a nice element of a bar chart. The mask can be the pixel array representing the scanned image of the Mona Lisa, and the ink can be polka dots, in which case you get whatever a Mona Lisa painted with polka dot ink would look like (presumably, a Mona Lisa with Chicken Pox). In any event these combinations point out the fact that either the ink or the mask, or both, can be defined by means of a pixel array. One can create an image of the Mona Lisa by using her scanned image pixel array as the ink, and depositing that ink through a mask which contains a large rectangular hole. Alternatively, one can create the same image by using black ink, and depositing that ink through a mask which is defined by her scanned image pixel array. It is absolutely essential that the reader maintain clearly in his mind at all times the difference between the uses of a pixel array in these two roles.

In the cases of gray ink or sampled black-and-white ink in which "0 is white" there is a special problem. If two images which are laid down at different times overlap, which one does one see in the area of overlap? Clearly the answer is the last one imaged. Unfortunately it may be more convenient for the printer to image objects in a different order from that visualized by the creator of the master. If the creator wishes to have his imaging order preserved he must tell the printer so by setting the imager variable *priority important* to a non-zero state. If images do not overlap or if ink is used in the "0 is transparent" mode then the order of laying down images is irrelevant. In this case the imager variable *priority important* should be set to the zero state so that the printer is free to choose its own image generating sequence.

7

Coordinate systems

It is necessary to introduce the concepts of *coordinate systems* and *transformations* between coordinate system into Interpress for a number of reasons. They permit the Interpress language to be device-independent. They permit the creator of a master to have great independence in how he chooses to measure positions on his desired output image, and in the units he chooses for these measurements. They also enable a simple means for the manipulation of masters created by others without having to make any changes to those masters. For example, through the use of transformations one can take two pages which were composed for portrait mode printing on an 8-1/2 by 11 inch page, and print them in a scaled down and rotated form in a side-by-side fashion on a landscape oriented 11 by 8-1/2 inch page. The entire process of such a two-up printing can be accomplished through the use of transformations without having to modify any of the Interpress description of the content of those pages.

A coordinate system used within Interpress is defined by establishing a pair of orthogonal axes, establishing their orientation in some framework, and, generally, establishing a unit of measurement which the coordinate system employs. In some instances we will deal with coordinate systems or entities which do not have associated physical units. That is, they are not measured in inches, meters, printers points, or any other physical units. Although it appears to be a contradiction in terms we will say that such a coordinate system or such an entity is *dimensionless*, meaning that it has no physical units of measurement associated with it.

Interpress deals with four different coordinate systems. These are designated the *Master Coordinate System* (MCS); the *Interpress Coordinate System* (ICS); the *Device Coordinate System* (DCS); and the *Standard Coordinate System* (SCS).

7.1 Master Coordinate System (MCS)

Interpress 82 considers any coordinate system used within the master to be a Master Coordinate System. The creator of a document is free to change the position, orientation, and scale factor of his MCS as often as he finds it convenient to do so. This facility permits the creator of a document to choose a coordinate system with orientation with respect to the page and with units which best suit his needs in describing the document in a manner that is independent of the ultimate printing device. This flexibility permits the MCS to be a dynamically changing coordinate system. Thus, when Interpress 82 speaks of the MCS it is not referring to a fixed coordinate system, but is talking about the current state of a coordinate system that may be dynamically changing. Experience has shown that it is easier to think about the master if one establishes an initial value for the Master Coordinate System, and then refers to other coordinate systems used within the master as being *local coordinate systems* which are related to this initial value. In this Reader's Guide we will refer to the initial coordinate system used by the Master as the Base Coordinate System, **BCS**, and refer to other coordinate systems in the master as being local coordinate systems related to the **BCS**. In these terms, then, all coordinate specifications of objects on the page are given in the **BCS** or in coordinate systems related to it by transformations of the type described below. A concatenation of all such transformations is used to generate the single transformation which converts coordinates in any local coordinate system used by the creator back to the **BCS**. Thus, a particular state of the **MCS** is always represented by a transformation which brings it back to the **BCS**.

7.2 Interpress Coordinate System (ICS)

The **ICS** is an intermediate coordinate system which is universally known to, and implicitly used in the communication of, all Masters. Its purpose is to decouple the arbitrarily chosen coordinate systems that the creator of the master uses in describing his images from the arbitrary coordinate system which each printer uses in its internal operation. Each creator of a master transmits within the master the transformation(s) which is/are required to convert from his coordinate system(s) to the Interpress Coordinate System. Each printer carries within its implementation the transformation which is required to convert from the Interpress Coordinate System to its Device Coordinate System (described in the following section).

The origin of the **ICS** is in the lower left-hand corner of a page which, when viewed in its normal viewing mode, has its long edge oriented vertically and short edge oriented horizontally. (Hereinafter a page printed with this orientation will be said to be printed in *portrait mode.* A page which, in its normal viewing mode, is rotated 90 degrees from such a page is said to be printed in *landscape mode.*) The unit of measurement in the **ICS** is the meter. Implicitly, all coordinates in the **BCS** are transformed into **ICS** coordinates for representation in the final master. Thus, the master can be expressed in a source independent and device independent coordinate system which is universally known by all printers produced by Xerox. The actual conversion into **ICS** coordinates may never explicitly occur. The Interpress master constructed by the creator contains the "recipe" and the "ingredients" for this conversion to be made, but it may never explicitly be displayed. (See the following Section on the Device Coordinate System).

7.3 Device Coordinate System (DCS)

The **DCS** is a coordinate system whose orientation and units are appropriate for the device on which the image is being created. The master supplies the transformations required to convert its coordinate systems into the **ICS** coordinate system. The printer supplies the transformation required to convert from the **ICS** coordinate system to the **DCS** coordinate system. All coordinates in the **ICS** are transformed into **DCS** coordinates for the page imaging process. The printer maintains a *current transformation* which represents the concatenation of all transformations from any local coordinate systems which the master may choose to create for specific local uses on a page to his **BCS**, and thence from **BCS** to **ICS**, and thence from **ICS** to **DCS**. Thus, original coordinates expressed in the creation of the master are transformed all the way into **DCS** coordinates by the current transformation during the generation of the final image at the imaging device. As described below this sequence of transformations is expressed as a single matrix which is the product of simple matrices. In the case of the type of matrices employed by Interpress 82, in the worst case, each such product involves twelve multiplications and eight additions. Similarly, in the worst case, the transformation of a set of coordinates from the **BCS** to the **DCS** by means of such a matrix involves four multiplications and four additions.

The ultimate transformation from the creator's local coordinate systems, back through to his **BCS**, thence to the **ICS**, and finally into the **DCS** is taken as the concatenation of all of these transformations. (See the Section 8 on Transformations.) In this case the creator's local coordinates are transformed all the way to device coordinates without ever having been materialized as Interpress coordinates. It is because of this that the **ICS** is only an implicitly utilized intermediate coordinate system.

Generally, the properties of the **DCS** are not known to the master creator. *However, if he does know its properties, and if he chooses to use an identical coordinate system with identical units the concatenation of all of the transformation matrices from his BCS to the DCS reduce to the identity matrix, and only the master's local transformations remain. If he uses no local transformations the transformation operations become the Identity transformation, and effectively disappear.*

7.4 Standard Coordinate System (SCS)

The Standard Coordinate System is a dimensionless coordinate system in which pixel arrays and character operators are defined. Its use will be described in later sections devoted to those subjects.

8 Transformations

Transformations are used to map coordinates measured in one coordinate system with its set of units to another coordinate system with its set of units. As described above a transformation is used to map from the **BCS** to the **ICS**, and another is used to map from the **ICS** to the **DCS**. The creator of the master may find it convenient to use local coordinate systems within a page that are different from the **BCS**. A trivial example is the translation of the origin of the **BCS** to a new location on the page. The use of such coordinate systems merely creates additional transformations which describe how to get from those coordinate systems back to the **BCS**. Thus, a coordinate specified in the master may need to be subjected to several transformations in order to map it all the way into the **DCS**. Fortunately, however, the effect of several transformations applied in sequence can be expressed as a single, combined, transformation.

A generalized representation of the coordinates of any point (x, y) in any coordinate system makes use of the homogeneous vector form, [x, y, 1]. This form is adopted so that a single matrix representation of all admissible transformations can be used.

(For those not familiar with the handling of matrices a description of the fundamental matrix operations utilized by Interpress is presented in Appendix D. It is recommended that even those familiar with matrix operations read Appendix D because of the somewhat different matrix methods used by Interpress.)

An admissible transformation in Interpress is represented by a 3x3 matrix, **M**, having the following structure:

$$\mathbf{M} = \begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

A matrix of the form **M** is used to transform the coordinates of the point (x_{from}, y_{from}) from one coordinate system to the point (x_{to}, y_{to}) in another according to the following matrix operation:

$$[x_{to} \ y_{to} \ 1] = [x_{from} \ y_{from} \ 1] * \mathbf{M}$$

(where * in this equation stands for Matrix multiplication)

In equation form this reduces to the following computations:

$$x_{to} = a*x_{from} + b*y_{from} + c$$
$$y_{to} = d*x_{from} + e*y_{from} + f$$

(where * in this equation stands for algebraic multiplication)

The most general form of Matrix **M** has the effect of performing a combination of non-uniform scaling, rotation, reflection about either of the axes, and translation. In Interpress specifically constrained forms of **M** provide for appropriate subsets of these generalized operations. These include:

1. Pure scaling in the same amount in both the x and y directions.

$$\mathbf{M} = \begin{matrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{matrix}$$

2. Scaling by different amounts in the x and y directions.

$$\mathbf{M} = \begin{matrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{matrix}$$

3. Pure rotation.

$$\mathbf{M} = \begin{matrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{matrix}$$

where a is a counterclockwise rotation of the coordinate axes.

4. Mirror Image about x-axis.

$$\mathbf{M} = \begin{matrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

5. Mirror Image about y-axis.

$$\mathbf{M} = \begin{matrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

6. Pure translation.

$$\mathbf{M} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{matrix}$$

Often two transformations are performed in sequence. The same result is achieved if the two matrices which represent these transformations are multiplied according to the mathematical rules for matrix multiplication *in the correct order*, and the resultant single product matrix applied to the original coordinates. The order must be maintained because matrix operations are not generally commutative. A typical instance of the use of matrix multiplication is the following:

The transformation **ID** represents the transformation from the Interpress Coordinate System to the Device Coordinate System. Thus,

$$[x_{Device}\ y_{Device}\ 1] = [x_{Interpress}\ y_{Interpress}\ 1] * \mathbf{ID}$$

The transformation **BI** represents the transformation from the Base Coordinate System to the Interpress Coordinate System. Thus,

$$[x_{Interpress}\ y_{Interpress}\ 1] = [x_{Master}\ y_{Master}\ 1] * \mathbf{BI}$$

Substituting for $[x_{Interpress}\ y_{Interpress}\ 1]$ from this second equation into the first we obtain:

$$[x_{Device}\ y_{Device}\ 1] = [x_{Master}\ y_{Master}\ 1] * \mathbf{BI} * \mathbf{ID}$$

This example demonstrates that the two sequential transformations which take us from **BCS** to **ICS** and then from **ICS** to **DCS** can by replaced by the single transformation **Q** which is obtained by performing the matrix multiplication **BI * ID**. Note that matrix multiplications are not generally commutative (i.e. interchangeable in the order in which they are executed) so that the matrix multiplication order **BI * ID** must be preserved. This example also demonstrates that the previously described transformations from the creator's local coordinate system, to the **BCS**, to the **ICS**, to the **DCS** is implemented by a series of matrix multiplications in which the matrices appear in the products in left-to-right sequence relative to the order of their occurrence. (See the portion of Appendix D titled *Interpress use of coordinate systems and transformations revisited* for a complete description of the left concatenation principle.)

The general principle that Interpress 82 adheres to with respect to the use of matrices is the following. The ultimate goal of all matrix operations is to produce the coordinates of images on the page expressed in the **DCS**. The way Interpress 82 achieves this is to start from the **DCS** with a series of coordinate system changes that leads to the **ICS**. These will certainly include a scale changing matrix, and may also involve translations, rotations and reflections. At each step of the way the creator views himself as being placed in the new coordinate system associated with that step. He then creates the matrix which will take coordinates expressed in this new coordinate system forward to the predecessor coordinate system in that sequence of steps. In a sense, then, the creator "backs" away from the **DCS** through a series of transformations, but as he takes each step in this process he defines the transformation matrix which will bring his coordinates forwards towards the **DCS**. This "backing out" process is accomplished by a series of left multiplications of matrices. That is, each new matrix is concatenated with all of its predecessors by a left multiplication operation. Note that in the preceding paragraph the matrix **ID** carries **ICS** coordinates into **DCS** coordinates, and the matrix **BI** carries **BCS** coordinates into **ICS**. Also note that the matrix BI left multiplies the matrix **ID**. If the creator of the master wishes to use a local coordinate system other than **BCS** (call it **LCS** for reference purposes) he thinks of himself as having stepped into that **LCS** coordinate system and writes the matrix **LB** that carries **LCS** coordinates into **BCS** coordinates. He then left concatenates that matrix on the existing matrix product to form **LB * BI * ID**. Note that while he moved *from* **BCS** *to* **LCS** he wrote the transformation which carries coordinates *from* **LCS** *to* **BCS**.

8.1 Initial transformations

At the beginning of each page the current transformation, denoted by T, is initialized by the printer to the Identity transformation, I, defined as:

$$I = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

The matrix **I** has the property that for any matrix **M**, **M*I = I*M = M**

At the beginning of each page the Interpress interpreter in the printer automatically concatenates the **ICS** to **DCS** transformation with the Identity transformation to create the transformation that maps the Interpress Coordinate System (**ICS**) to the Device Coordinate System (**DCS**). This is achieved by setting:

T := ID * I = ID

Generally, although not necessarily, the first operation performed for each page by the master is to concatenate the **MCS** to **ICS** transformation with the value of T created by the printer. This operation can only be accomplished if each page explicitly invokes the required transformations. Such invoking may be implemented by having the preamble write and store in the page frame a composed operator whose execution carries out the required transformation. This composed operator can then be executed as the first operation of every page. This results in the value for T which will carry master coordinates into device coordinates, namely:

T:= MI * ID

If the master creator and the printer have the same coordinate system and units of measurement the transformation, **T**, is the identity transformation

$$T = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

which is represented by the rational sequence 1, 0, 0, 0, 1, 0.

8.2 Transformations used in conjunction with masks

A mask of any type is subjected to the current transformation each time that it is used. The current transformation may change a mask in a wide variety of ways. In their most common use transformations are applied to:

1. Translate the mask with respect to the page image.
2. Rotate the mask with respect to the page image.
3. Change the size of the mask. That is, change the size of the hole in the stencil through which ink will be deposited. The length and width dimensions of the hole may be independently changed in size so that the aspect ratio of the shape represented by the hole may be changed. Thus, for example, a square can be changed to a rectangle, or the height of a letter can be changed proportionately more than its width. Some printers might only

support limited subsets of this general capability. In particular, those that use pixel array representations of character masks will generally not contain such representations for characters which have been subjected to such transformations.

Masks may be defined by the creator of the master, or they may be defined by sources outside of the master. If the creator of the master defines a mask he does so within his coordinate system, and has complete control over it. He can define transformations which modify the size and position of the mask for each time that it is invoked, and include those transformations in his master in such a way that each becomes part of the current transformation at the time the mask is used at the printer. He knows precisely what the effect of his current transformation will be on the mask for each of these invocations. Thus, the creator could take the same mask and cause the shape it defines to be printed on a page with different orientations, and with different scalings in each orientation. The most common use of a transformation in conjunction with a mask is to position the mask at different locations on the page so that the same image may be deposited at different locations.

In some instances masks which the creator of a master wishes to use are defined by others. The two most typical instances of this are the masks associated with standard character fonts, and the masks defined by pixel arrays generated by input scanning devices or computer graphic programs. We will deal with the transformations associated with such masks in the following paragraphs.

8.3 Transformations used in conjunction with character operators

A character is imaged by a special composed operator called a *character operator.* We will discuss the details of character operators at a later point in this document. At this point we will focus our attention on the mask associated with a character operator, and on the role transformations play in the modification of such masks. This role is both critical and subtle. Understanding it is the key to understanding the use of character operators.

The mask for a character operator is defined in a Standard Coordinate System which is dimensionless. Each character mask has its origin at the origin of the SCS, its y-axis vertical and its x-axis horizontal when the character is viewed in the normal viewing position. The character is defined within a rectangle which is one *unit* high. Think of the unit as just that, a unit. It is not one inch, one printer's point, one foot, one meter, or any other dimension. The unit height rectangle has the characteristic that if such a mask is abutted below itself it is in precisely a correct position relative to the next line of text. From a traditional printing viewpoint the rectangle is exactly analogous to the "point size" of the type. The creator of the master brings such a mask into his coordinate system by applying a transformation which scales this unit high rectangle into a rectangle of the height of the point size which he wishes to use, expressed in the units of his coordinate system. For example, if he chooses to use printer's points as the units of his coordinate system, and wishes to obtain a 10-point character mask he scales by the factor 10. This takes the dimensionless unit high rectangle into one that is 10 units high in his coordinate system, and thus he obtains a 10-point mask since his unit is one point. On the other hand, if the units of his coordinate system are inches he would apply a scale factor of 10/72. This takes the dimensionless unit high rectangle into one that is 10/72 units high in his coordinate system, and he obtains a 10-point mask because 10/72 inches is equal to 10 points.

From the viewpoint of the creator of the master the character mask which he has transformed into his coordinate system is in the normal viewing position with its y-axis aligned with his y-axis. If he desires to print a page whose characters are rotated so that their y-axes are not aligned with his he must supply a rotation as well as a scaling in the transformation which brings the character mask into his coordinate system.

Once the character mask is in the master creator's coordinate system it may be subjected to subsequent transformations in a fashion which is identical to those which he applies to masks which he himself creates. As before, the creator of the master has complete control over subsequent uses of the mask. He can define transformations which modify the size and position of the mask for each time that it is invoked, and include those transformations in his master in such a way that each becomes part of the current transformation at the time the mask is used at the printer. He knows precisely what the effect of his current transformation will be on the mask for each of these invocations. Thus, the creator could take the same mask and cause the character mask to be imaged on a page with different orientations, and with different scalings in each orientation. Thus, although he might create a 10-point, portrait-oriented character mask with the transformation which brings the character into his coordinate system, he can change the point size and orientation of that mask by means of further transformations as he uses it to image different instances of the same character on the page.

8.4 Transformations used in conjunction with pixel arrays

Pixel arrays representing images are generally created by scanners or by computer graphics generating programs. We will speak of the initial scan direction as the *fast scan* direction, and the scan direction orthogonal to it as the *slow scan* direction. The creator of a master thinks of a pixel as a dimensionless entity, occupying a square area one unit on a side. For imaging purposes this unit area will be scaled to the size required to bring the complete image to its desired size. (Note that this concept works even if one were considering skywriting. If one wanted an image one mile high of an 8 x 10 photograph, scanned with 3000 pixels along the 10 inch directon and 2400 pixels along the 8 inch direction, each pixel in the image would be scaled to be 1.76 feet on a side.)

The creator of a master considers the pixel array to be in the Standard Coordinate System (SCS) with its y-axis pointing up and its x-axis pointing to the right. The first pixel to appear in the scanned information is located at the origin. Subsequent pixels along the first scan line are in the positive y-direction, i.e. the fast scan direction is in the positive y-direction. Subsequent scan lines proceed in the positive x-direction, i.e. the slow scan direction is in the positive x-direction. Assuming a rectangular pixel array of dimension y pixels along the fast scan direction by x pixels along the slow scan direction, the corners of the array are located at (0,0), (0,y), (x,y), and (x,0). Note that this orientation in the SCS is determined completely by the starting position and direction of the scanning process, and is completely independent of the normal viewing orientation of the image being created in a raster format. The image as viewed in this coordinate system could be on its side, upside down, mirrored, and so on, depending on the starting position and initial direction of scan. For example, if scanning starts in the upper left hand corner, fast scan top to bottom, slow scan left to right the image in this coordinate system would appear to be upside down and backwards. (See Figure D-2 in Appendix D.)

The creator of the master provides a transformation which brings the pixel array into his coordinate system and normalizes it. A pixel array is considered to be normalized in the Base Coordinate System when it has the following properties:

1. The pixel array is oriented in a meaningful manner. That is, it appears "upright" in the **BCS.**
2. The lower left hand corner of the array is located at the origin of the **BCS.**
3. Assuming a rectangular pixel array y pixels high by x pixels wide, the corners of a normalized array are located at (0,0), (0,1), (x/y,1), and (x/y,0). Note that this normalized form is a unit high rectangle, independent of the number of pixels along the y-direction.

To accomplish this normalization the creator of the master must know the following facts about the raster scan process which was used to create the pixel array:

1. The number of pixels in the array along the x- and y-directions in the array's representation in the **SCS.**
2. The rotation, translation, and/or reflection relationship from the **SCS** to his **BCS.** This is equivalent to his knowing the starting point and directions of the fast and slow scans.

The transformation which brings the pixel array from the **SCS** to his **BCS** contains a scaling by 1/xpixels or 1/ypixels, a rotation by the amount required to bring the **SCS** orientation to his **BCS** orientation, possibly reflections about one (or both) of the axes, and a translation to bring the lower left hand corner of the array to the origin of his **BCS.** Once this transformation has been established the normalized pixel array can be scaled, rotated, and translated to any size, position, and orientation the master creator desires. It will then be subjected to the current transformation whenever it is imaged. (See Appendix D for a detailed example.)

All of this theory must be understood and applied in creating masters. As a practical matter, however, most raster scan printers will not have the capability to change the order in which pixels are presented to the printer from that in which they were created by the scanner. To do so generally requires the relocation of several million bits of information within a memory system, and the time to perform that task is generally not compatible with printing speed requirements. Therefore, the final transformation which is in effect at the time the pixel array is imaged must generally be such that the pixels are presented to the printer in the same order as that in which they were created by the scanner. The creator of the pixel array must have foreknowledge of the image generating scan sequence of the printer relative to the page orientation, and must arrange his scan conversion process to match it appropriately. Thus, in general, it is not possible to do page rotation and two-up printing for pages which contain scanned images. Interpress provides the concept of *easy net transformations* to describe the set of transformations which result in images which the printer can handle readily. Most printers will reject pages which contain images whose associated transformations are not in the *easy* set.

9

Reference key

The first eight sections of this document presented a general overview of the Interpress 82 language, and of its major features and constructs. An understanding of these sections will significantly reduce the difficulty of the task of understanding the Interpress 82 language specification. There still remains within the presentation of the language a number of subtle and complex points which are difficult to grasp because of the concise form of presentation. Section 9 of the *Interpress 82 Reader's Guide* provides material which substantially amplifies and explains these points.

Note: The paragraph numbers in this section are in increasing magnitude, but there are gaps in their sequence. For ease of reference, the paragraphs in this section of the Reader's Guide are numbered in one-to-one correspondence with their correlated sections in the *Interpress 82 Electronic Printing Standard.* The gaps in paragraph numbering occur because not every paragraph in the *Interpress 82 Electronic Printing Standard* is commented upon in this section.

2.2 Types and literals

In the description of types in this section the statement is made "There are six types in the base language: Number, Identifier, Mark, Vector, Body, and Operator." This appears to be in contradiction with the number of types presented in Part I of this Reader's Guide where a larger number of types were enumerated. The apparent contradiction stems from the use of the phrase "base language". Note that Chapter Two of the Interpress 82 language specification is devoted to "The Base Language". It is in the context of the Base Language defined in Chapter Two that there are six types. In the context of the entire document there are twelve types.

2.2.1 Numbers and integers

The use of the "/" character to separate the numerator and denominator of a rational is a convention of the presentation of information in the Interpress language specification document. The "/" character is not used in the encoding scheme as part of the mechanism for transmitting a rational number.

2.2.3 Marks

A *Mark* can only be removed from the stack by a *matching* UNMARK operator, i.e., one executed in the same context, or during a mark recovery. Information which will help clarify the definition of a *matching* mark is to be found in Part I of this Reader's Guide under the section titled The frame.

2.2.4 Vectors

In the definition of the *lower bound l* and the *upper bound u* of the indices of a vector the statement is made that "l must be less than or equal to $u+1$." There appears to be an inconsistency here because if, for example, $u=5$ then l could be as large as 6. In fact, it is the convention that a *null* vector is defined in precisely this fashion, with its lower bound one greater than its upper bound. Note that if, in the above example, both $u=5$ and $l=5$ then the vector has precisely one component, designated by the index value 5.

2.2.5 Bodies and operators

The examples in the "bullets" in fine print near the end of this section are difficult to absorb at this point since the syntax and semantics of the language have not yet been defined. For clarity of exposition at this point we will repeat and explain two of the examples here. This explanation will also serve as an introduction to the syntax and semantics of the language.

Example 1

"Conditional execution is provided by the IF and IFELSE operators, which take a Body and an Integer as arguments, and execute the Body if the Integer is non-zero. E.g. 2 FGET 3 GT {conditional body} IF."

The execution of this example proceeds in the following fashion:

Operator or Operand	Stack	Result
2	2	Push the value 2 onto the stack.
FGET	(Frame(2))	Pop the value 2 from the top of the stack and use it as an index into the current frame. Retrieve the value from location 2 in the current frame and push it onto the stack.
3	3 (Frame(2))	Push the value 3 onto the stack. The stack now contains the value 3 followed by the value obtained from location 2 in the current frame.

GT	0 or 1	Pop the top two values off of the stack. If the value obtained from location 2 in the frame (which must be of type number) is greater than the value 3 push a value 1 onto the stack, else push a value 0 onto the stack.
{conditional body}	{cond'l body} 0 or 1	Push the {conditional body} onto the stack.
IF		Pop the {conditional body} off of the stack, pop the result, 0 or 1, of the execution of the GT operator off of the stack. If that value is not equal to 0 execute the {conditional body}, otherwise do nothing.

Example 2

```
{--draw a solid box with size given by the top two stack values--

    TRANS
    0 0 4 2 ROLL
    MASKRECTANGLE

}MAKESIMPLECO
```

This procedure (composed operator) requires that the width w and height h of the desired rectangle be on the stack before it is invoked.

Operator or Operand	Stack	Result
	h w	
TRANS	h w	Moves the origin (0,0) of the MCS to the current position in the image. This locates a corner of the rectangle at the current position in the image. The stack is unaffected.
0 0 4 2	2 4 0 0 h w	The 0, 0, 4, 2 are pushed onto the stack, causing the stack to contain 2, 4, 0, 0, h, w.

ROLL	h w 0 0	The 2 and 4 are popped off the stack by the roll operator which then proceeds to "roll" the top 4 elements on the stack until the bottom 2 of the top 4 elements become the top 2 elements of the stack.
MASKRECTANGLE		This draws a rectangle of width w and height h with lower left hand corner at (0,0).

2.4.3 Vector operators

This note is just a reminder as to the meaning of an asterisk, "*", preceding an operator. The *GET operator is the first instance of such a use in the Interpress language specification. The asterisk signifies that this operator has been introduced into the description of the language solely for the purpose of simplifying the succeeding explanations of other operators. The *GET operator is not a primitive operator in the Interpress language. It cannot be used within a master. There is no mechanism for transmitting this operator in the Interpress encoding system.

2.4.7 Control operators

The fine print examples under the IFELSE operators are further clarified below.

In the text it states that the effect of the conventional computer language statement "**if** *i* **then** B1 **else** B2" is obtained with the Interpress sequence "*i* B1 IFELSE B2 IF". The following is the expansion of the Interpress sequence.

Operator or Operand	Stack	Result
i	*i*	Pushes the value *i* onto the stack.
B1	B1 *i*	Pushes the body B1 onto the stack.
IFELSE	0 (if *i* not equal 0) or 1 (if *i*=0)	Pops B1 and *i* off of the stack. If *i* is not equal to 0 it executes B1, and pushes a 0 back onto the stack. If *i*=0 it does not execute B1 and pushes a 1 back onto the stack.
B2	B2 1 or 0 (depending on *i*)	Pushes B2 onto the stack.
IF		Pops B2 and 1 or 0 value pushed by ifelse. Executes B2 if that value was not a zero. Does not execute B2 if that value was a zero.

The second example in the text reads "*i*1 B1 IFELSE { *i*2 B2 IFELSE B3 IF} IF". This form follows from the previous one where the body B2 in the previous example is replaced by the body enclosed in the "{" and "}" in this one. The first IFELSE either does or does not execute B1 according to the value of *i*1, and leaves a value of 0 or 1 on the stack. The last IF then either does not or does execute the body within the "{" and "}" according to the value of 0 or 1 left on the stack by the first IFELSE. If it does cause the body enclosed in the "{" and "}" to be executed we find another instance of the previous example inside the "{" and "}".

The description of the IFCOPY operator is somewhat confusing because it does not make clear that the *testCopy* operator, which is an input parameter to the IFCOPY operator, is itself a composed operator that must have previously been created in the master. The IFCOPY operator description states that *testCopy* is called with the copy number (an Integer) and a copy name (an Identifier). Neither the copy number nor the copy name are defined within Interpress 82. This is another example of the concept introduced in Part I of this Reader's Guide of the *unalterable state*, or *environment*, in which the Interpress interpreter is embedded in a given printer. (See the section titled State in Part I.) When IFCOPY is executed the Interpress interpreter in the printer essentially leaves the Interpress domain, and enters a special procedure which is supplied to the printer outside of the Interpress domain. The entire mechanism of the *testCopy* operator is left open to the installation which uses Interpress 82. The only requirement imposed by Interpress 82 is that the *testCopy* composed operator return a single integer. If the value of this integer is zero the body operand of the IFCOPY operator is not executed. If the value of this integer is not zero the body operand of the IFCOPY operator is executed. In a typical application environment the printer might be supplied with a list of document recipients together with the number of copies each is to receive. The *COPYNAMEANDNUMBER operator alluded to in the description of the IFCOPY operator could be implemented so that different copies of the document may be defined by a two-level characterization which includes a receiver's name, and within name, a copy number. Thus, a copy might be designated as Accounting, Copy 3; or Receiving, Copy 2. The printer maintains a record of which copy name and copy number it is currently working on. (This record is maintained outside of Interpress 82.) The IFCOPY operator invokes a procedure in the printer, and outside the Interpress language domain, which supplies the copy name and copy number to the IFCOPY operator. The IFCOPY operator, in turn, supplies these values as parameters to the *testCopy* composed operator. The *testCopy* composed operator returns a 1 or a 0 to the IFCOPY operator which then proceeds to cause, or skip, the execution of its associated body operand.

3.1 The skeleton

Note the following three facts about the preamble.

1. It is a body which has precisely the same structure as any page body in the master.
2. It has a special significance as being a preamble only by virtue of its being the first body in the master.
3. It is not permitted to produce any output to the page image, nor leave any information on the stack.

Because it is a body like a page body it is subject to the same constraints as those applied to any other page body in the master. Thus, it is executed in a DOSAVEALL fashion. As a result it cannot affect the imager state that is seen by any other page body in the master. If the creator of the master wishes every page body in the master to work with an Imager State that is different from the one initialized by the system he must take special steps to establish that state. The simplest way to accomplish this is to create a composed operator in the preamble whose execution will modify the Imager State to the desired condition. This composed operator is stored in the page frame that is given to every page body. The master can then cause each page body to begin with the execution of this composed operator.

3.2 Environments and names

We have already noted that the IFCOPY operator is a link to the outside world in addition to the FINDFONT and FINDDECOMPRESSSOR operators and the sequenceInsertFile encoding-notation.

3.2.2 Hierarchical names

The description of a hierarchical naming system is generally clear. The one element which might lead to confusion is the use of the "/" character in the descriptive text. It appears repeatedly in forms such as Xerox/, Xerox/Helvetica/. . ., Xerox/TimesRoman/ . . ., and Mergenthaler/TimesRoman/. . .. the use of the "/" in this context is intended to convey the concept of hierarchical structure. It has the meaning that the value identified by a name such as Xerox/TimesRoman/Italics is to be obtained by locating an element identified by the name Xerox, within that element locate an element identified by the name TimesRoman, and within that element locate an element identified by the name Italics. The "/" is not a legitimate character in an identifier which can consist only of the lower case letters of the alphabet, the digits 0 through 9, and the minus (-) sign. The "/" does not itself appear in the hierarchical set of identifiers which would be presented in a sequence of tokens in the encoding system.

4.1.1 Priority

In the first paragraph of this section it states that "laying down an object obscures any overlapping parts of objects that have been previously placed on the page image unless the color is transparent." If the color is transparent the portions of the ink defined by 0's in the pixel array are treated as transparent in the sense that they cause no ink to be applied to the page. If a first object is imaged with the same ink as the second object the statement still holds true. However, if the first object was imaged with a different ink from the second object it probably will not remain true. The 0's in the pixel array defining the ink for the second object may coincide with pixels in the first object which contain ink, and the transparency parameter will allow those pixels of the first object to remain unaltered. Thus, although the second object may overlap the first object it will not necessarily obscure that first object if the first object was imaged with a different ink.

4.2 Imager state

This section states that "The *SETMEDIUM operator, called at the beginning of a page, alters the imager state to reflect the details of the medium selected for printing the page." It should be noted that the calling of the *SETMEDIUM operator is implicit in the printer's implementation. It is an asterisked operator which cannot be invoked in a master. The creator of the master can assume that this operator is automatically executed at the beginning of each page body. Note that the setting of the medium defines the normal orientation of a page from the printer. A setting in which the medium dimension in the y-direction of the Device Coordinate System is greater than the medium dimension in the x-direction produces a portrait page in the normal orientation of a page from the printer. Conversely, a setting in which the relative size of the medium's dimensions along the two axes is reversed produces a landscape page in the normal orientation of a page from the printer. It is recommended that the portrait orientation be the conventional one if the printer provides that option in its implementation.

4.3 Device coordinate system

The concepts underlying the *DROUND operator are subtle, and require clarification. This operator takes the coordinates (x,y) of a point in the Device Coordinate System as its operands and produces the coordinates of another point (X,Y) in the Device Coordinate System. The coordinates provided as operands to the *DROUND operator could come from anywhere. However, in most applications they will have been created in the master and converted to the DCS by means of the current transformation. The calculations inherent in this transformation will generally produce values for (x,y) which are non-integral in the DCS, and therefore the point (x,y) will lie between the grid points at which the device can actually lay down pixels. The *DROUND operation produces the device coordinates (X,Y) of the grid point best representing the point (x,y). The subtlety in this last statement lies in the phrase "best representing the point (x,y)". The question *DROUND addresses is the selection of the best grid point to use relative to the actual point (x,y). At first glance this appears to be a simple round-off question. Merely round-off x and y to their nearest integer values to obtain X and Y. Unfortunately this simple algorithm might not result in the selection of the grid point best representing the point (x,y). There are many factors which influence that selection, as a function of the characteristics of the printer. Each printer has its own imaging "footprint" whose characteristics are a function of many parameters. If we use a laser driven Xerographic imaging process as an example these parameters might include:

- The wobble in its scanning mechanism
- The optics of the system
- The xerographic properties of the imaging surface
- The properties of the ink depositing mechanism
- The properties of the ink transferring mechanism
- The properties of the ink fusing mechanism

The combination of all of these properties causes the visible pixel finally reproduced on the paper to have a characteristic shape which we have called its footprint. The shape of the footprint determines the grid point best representing the point (x,y). It is based on this shape for a particular printer that the *DROUND function is built for that printer. It may be different for each class of printer.

4.4.2 Notation

This section provides purely mathematical definitions for the concepts of "point" and "vector" transformations in the following terms:

"Point" transformation: T_p(x,y,m)=(X,Y), where [X,Y,1]=[x,y,1] *m*.
"Vector" transformation: T_v(x,y,m)=(X,Y), where [X,Y,0]=[x,y,0] *m*.

The definition of a point transfomation is straightforward and should be clear. It does precisely what we defined transformations to do, namely provides the coordinates of the point (X,Y) as expressed in one coordinate system in terms of the coordinates (x,y) of the same point as expressed in a second coordinate system where the matrix *m* represents the transformation from the second coordinate system to the first. The effect of the vector transformation is not quite so clear. It is, perhaps, more easily understood if we carry out the matrix operations and write them in equation form.

Point Transformation:

$$[x,y,1]\ m = [x\ \ y\ \ 1] \ * \begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

$$\begin{aligned} X &= ax + by + c \\ Y &= dx + ey + f \end{aligned}$$

Vector Transformation:

$$[x,y,0]\ m = [x\ \ y\ \ 0] \ * \begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

$$\begin{aligned} X &= ax + by \\ Y &= dx + ey \end{aligned}$$

In these forms it is clear that the translation terms, c and f, are missing from the vector form of transformation. Thus, the vector transformation behaves as though the two coordinate systems had a common origin. It answers the question, "If I had a physical vector in the second coordinate system whose components were of magnitude x and y, what would be the magnitude, X and Y, of its components in the first coordinate system, given the transformation *m* from the second coordinate system to the first?"

In the discussion of ROTATE the statement is made "The rotation can be viewed in two ways: it will rotate the coordinate axes *counterclockwise* by the angle a, while it will rotate geometrical figures *clockwise* by the angle a." This statement may be confusing to those not thoroughly familiar with transformation theory. Any transformation may be viewed from either of two equally valid viewpoints which are referred to as the *alias* and the *alibi* viewpoint. In the alias viewpoint the points of the plane are fixed and two different coordinate frameworks are set down in it. The transformation is viewed as describing how to obtain the coordinates of a point in one coordinate system from its coordinates in the other. In this view each point has two identities, one as seen from the first coordinate framework, the other as seen from the second coordinate framework. It is this viewpoint which is adopted in the description of Interpress. From this alias viewpoint a rotation results in the coordinate system rotating counterclockwise by the angle *a* relative to all the points in the stationary plane.

From the alibi viewpoint the coordinate framework stays fixed and each point in the plane is moved from a first location to a second location. The transformation is viewed as describing how to obtain the new location of a point from its old location. From an alibi viewpoint a rotation results in the points of the plane appearing to rotate in a clockwise rotation relative to a stationary coordinate system. From this viewpoint, then, it is the geometric figure represented by a collection of these points which appears to rotate clockwise by the angle a in the plane.

4.4.5 The current transformation

This comment is a repeat of a point made in Part I of this Reader's Guide. At the beginning of each page the current transformation, denoted by **T**, is initialized by the printer to the Identity transformation, **I**, defined as:

$$I = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

The matrix I has the property that for any matrix **M**, **M*I = I*M = M**

At the beginning of each page the Interpress interpreter in the printer automatically concatenates the **ICS** to **DCS** transformation with the Identity transformation to create the transformation that maps the Interpress Coordinate System (**ICS**) to the Device Coordinate System (**DCS**). This is achieved by setting:

T := ID * I = ID

Generally, although not necessarily, the first operation performed for each page by the master is to concatenate the **MCS** to **ICS** transformation with the value of T created by the printer. This operation can only be accomplished if each page explicitly invokes the required transformations. Such invoking may be implemented by having the preamble write and store in the page frame a composed operator whose execution carries out the required transformation. This composed operator can then be executed as the first operation of every page. This results in the value for **T** which will carry master coordinates into device coordinates, namely:

T:= MI * ID

If the master creator and the printer have the same coordinate system and units of measurement the transformation, T, created by **T:= MI*ID**, is the identity transformation

$$I = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

which is represented by the rational sequence 1, 0, 0, 0, 1, 0.

4.4.6 Instancing

This section describes only a portion of the total process of creating the image of a character on a page. It deals with the identification of the character's "shape" and an initial establishment of its origin at the "current position". (The remaining portions of the total process are described in Sections 4.9, 4.9.1, and 4.9.2. In fact, these sections should be read in this Reader's Guide prior to the reading of this section 4.4.6.)

The imaging of a set of characters is effected by the operator SHOW which takes as an operand the vector *v*. That is, the operator SHOW must find a vector, *v*, on the top of the stack at the time it is called.

The vector *v* contains a string of characters. This string of characters will be imaged beginning at the current position as a result of the execution of the SHOW operator. The results achieved by executing the SHOW operator are stated to be:

> **where** for each element *e* in *v*, beginning with *v*'s *l* and ending with *v*'s *u* (i.e. for all elements of *v* from its first through its last) perform
>
> {TRANS *showVec* IGET *e* *GET DO} DOSAVESIMPLEBODY.

For each element *e* of the vector *v* the SHOW operator causes a set of operations to be performed as described in the following paragraphs. We will do so by describing each of the operands and operators in the above sequence.

The TRANS operator sets the origin of the current transformation *T* to the rounded current position. This is performed because, by convention, each character is structured with its origin at (0,0). Thus, the current transformation is conditioned so that the character will be laid down with its origin at the rounded current position. (See the comments on Section 4.3.5). The selection of the actual character which will be placed at that position is the result of a complex and subtle series of operations of which the SHOW operator is only a portion.

The Imager Pool variable with the name *showVec* (Index value = 12) contains a font whose imaging operators have been scaled to a specific size. For the purposes of this explanation let us assume that it is a 10-point Helvetica portrait font. (See 4.9.1, which describes the process of locating a font, and 4.9.2 for a description of the MODIFYFONT which establishes this size, and SETFONT which places the font into the Imager Pool variable *showVec*.)

The operator IGET brings the font from the Imager Pool variable *showVec* to the stack.

The operand *e* is a typical character code extracted from the vector *v*.

The operator *GET uses the value of *e* as an index into the font on the stack to obtain the proper imaging operator from the font for the imaging of the character element identified by the character code *e*.

The operator DO causes that imaging operator to be executed.

The whole sequence is executed as a body by the operator DOSAVESIMPLEBODY. Because this last operator is of the "DOSAVE" form all of the states of the system, other than the states of the persistent Imager Pool, are restored at the end of the imaging of each character.

Now, all of this seems straightforward and unsubtle. It would appear that the result of the SHOW operator would produce the imaging of the string of characters whose codes are contained within the vector *v* in a 10-point Helvetica portrait font. Such is not necessarily the case. There is a hidden facet to the imaging task contained within the definition of a font. Each element of a font is an *imaging operator* which will create the image of the character with which it is associated. Because such an element is an *operator* it may perform a number of functions before it actually lays down the image of its associated character. Sections 4.9.1 and 4.9.2 describe the process of locating a font, modifying its size, storing it in the Imager Pool variable named *showVec*, and the processes of executing a font character operator. They must be read and understood before the process of imaging the characters of the vector *v* can be completed. In anticipation of that process let it be stated here that as a consequence of the present *state* of the system the execution of the character operators of the 10-point Helvetica portrait font currently held in the Imager Pool variable *showVec* might actually, for example, result in the imaging of a set of 6-point Helvetica landscape characters on the page. Many other alternatives are also possible.

4.5 Current position operators

The reader should take careful note of the fact that all input operands associated with the current position operators are expressed in the Master Coordinate System. They are transformed by the point or vector forms of the current transformation into the Device Coordinate System where they affect the values of the current position held in the Device Coordinate Variables, DCScpx and DCScpy. Because of this transformation process a change in the x-direction in the MCS may result in an actual change in the y-direction in DCS, and conversely a change in the y-direction in MCS may result in an actual change in the x-direction in DCS. Thus, where the definition of SETXREL states "i.e. a relative displacement in the x-direction is added to the current position" that relative displacement is referenced to the MCS before the application of the current transformation, and may actually result in a change in the y-direction in the DCS. The same type of statement holds for the SETYREL operator.

Note that the GETCP operator takes the current position in the DCS and converts it into the coordinate framework of the MCS. This requires the inverting of the current transformation matrix. It is in reference to this inverting operation that the phrase "T must be well enough conditioned to invert" applies. The following material explains that phrase.

The inverse of the current transformation *T* expressed by:

$$T = \begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

is given by the transformation matrix:

$$T^{-1} = \begin{matrix} \frac{e}{ae - bd} & \frac{-d}{ae - bd} & 0 \\ \frac{-b}{ae - bd} & \frac{a}{ae - bd} & 0 \\ \frac{bf - ec}{ae - bd} & \frac{cd - af}{ae - bd} & 1 \end{matrix}$$

which only exists if $ae - bd <> 0$.

Note that even if $ae - bd <> 0$, its value may be so small that the quotients in the above matrix exceed the computational capacity of the printer. A matrix which has that property is not well enough conditioned to invert.

4.6 Pixel arrays

Pixel arrays are covered in detail in the section titled "Transformations Used in Conjunction with Pixel Arrays" in Part I of this Reader's Guide. That material in Part I should be reviewed carefully if the interpretation of the section on Pixel Arrays proves difficult. This section of Part II will be devoted to the finer technical details of pixel arrays as they are presented in Section 4.6 of the Interpress Language Specification.

The last paragraph of Section 4.6 introduces the concept of a *net transformation* which carries a pixel array into the Interpress Coordinate System. The purpose of a net transformation is to provide a method for discussing a pixel array within Interpress that is independent of the Master Coordinate System and the Device Coordinate System. This choice is somewhat confusing since the Interpress Coordinate System is generally not materialized within any processing operations. As described in the section titled Transformations Used in Conjunction With Pixel Arrays in Part I of this document it is much clearer, and more convenient, to define a pixel array in the Standard Coordinate System, and then transform it into the Master Coordinate System, where a pixel acquires the dimensional units of that system. It is then subjected to the current transformation T at the time it is used in an imaging operation, in which it may act either as a mask or as sampled black-and-white ink. Since T takes the pixel array all the way to device coordinates the net transformation (to Interpress coordinates) is never materialized within the printer. The transformations associated with these operations are described in Part I in the section titled Transformations Used in Conjunction With Pixel Arrays.

It is clear from this that the *net* transformation will never be materialized within any specific implementation. The concept of a net transformation is a convenience which permits each printer to describe to creators of masters the transformations which it can handle easily. This description is presented in terms of the universally used ICS. Each printer provides a list of transformations, expressed at the ICS level, that it considers *easy*. This list provides the creator of a master with a statement of *easiness* that is independent of the inner details of the printer implementation. The creator of a master can readily determine whether or not he is creating a transformation which, if it were materialized at the ICS level, is or is not a member of a printer's *easy* set. In any implementation the transformation associated with a pixel array will probably never be materialized at the ICS level within the printer, but will be taken all the way to the DCS level. Thus, a printer will examine whether or not a specific transformation is a member of its *easy set* by examining that transformation in the DCS domain.

As described above, the *net* transformation is generally not materialized in any printer, nor is such a transformation examined by the printer. The *net* transformation described in the last paragraph of Section 4.6 is stated in the following manner. The pixel array is defined by <*xpixels*, *ypixels* 1 1 1 *m samples* MAKEPIXELARRAY>. In this definition *xpixels* and *ypixels* are the number of pixels in the (rectangular) array along the x- and y-axes, respectively. The parameters 1, 1, and 1 are constant place holders for parameters reserved for future use. *m* is the transformation which carries the pixel array from the Standard Coordinate System to the Master Coordinate System. *samples* is the vector of pixels, where each pixel is a single bit. Pixels are presented in scan sequence within the *samples vector* under the assumption that they have been generated by a sequential scanning process. The *net* transformation, designated by *nm*, is given as *nm*=<*m um* CONCAT D^{-1} CONCAT>, where D^{-1} is the inverse of the ICS-to-DCS transformation. Here, again, the *m* transformation is the one used in the definition of the pixel array as stated above. The transformation *um* contained within the definition of *nm* is the value of the current transformation at the time the pixel array is used in a masking operation. It carries the pixel array from the MCS to the DCS. Thus, the portion of the definition of *nm* which reads *m um* CONCAT produces the transformation which carries the pixel array from the Standard Coordinate System in which it was originally defined to the Master Coordinate System, and then all the way to the Device Coordinate System. Since the goal of the *net* transformation is to present the pixel array in the Interpress Coordinate System it is now necessary to bring it back from the Device Coordinate System to the Interpress Coordinate System. The transformation which will do this is the inverse of the transformation which carries the Interpress Coordinate System to the Device Coordinate System. If the transformation from ICS to DCS is designated as D, the transformation from DCS to ICS is designated as D^{-1}. Thus, the transformation represented by *m um* CONCAT must itself be concatenated with D^{-1}, and that concatenation is precisely what is expressed by the final D^{-1} CONCAT in the definition of *nm*. In point of fact, it is impossible for the master to generate D^{-1}, so that the master could never generate *nm* by a process equivalent to the one used to define *nm* in the Interpress 82 standard. A master could generate *nm* directly by keeping track of its MCS to ICS transformation, and thereby avoid the necessity to know or generate D^{-1}. As noted above, such a generation is not necessary since a printer will examine the transformation associated with a pixel array in its own DCS to determine if it is *easy* or not.

4.7 Color

Color is covered in detail in Chapter 6, "The Imaging Model" in Part I of this Reader's Guide. That material in Part I should be reviewed carefully if the interpretation of the section on color proves difficult. This section of Part II will be devoted to the finer technical details of color as they are presented in Section 4.7 of the Interpress Language Specification.

The second paragraph is somewhat misleading. It would be better stated in the following terms. Constant colors are limited to black and shades of gray, specified by the light intensity absorbed by the image as a fraction *f* of the light intensity that is incident. (Note that most printer implementations contain only black inks, hence cannot truly represent a gray ink in the literal sense of the word. Gray inks in such printers are simulated by regular patterns of black and white pixels which are organized so that the human eye integrates them to the desired shade of gray.) There is a second kind of color called *sampled black-and-white* color which is defined by a pixel array in the manner defined below. A type *Color* records a color description: either a constant color or a sampled black-and-white color. In the case of sampled black-and-white color there is a mode in which the "white" portion of the sample is treated as *transparent.* This latter case is described in the following paragraphs.

In Section 4.7 the statement is made "This region is *tiled* repetitively to build an arbitrarily large pattern of color which encompasses the entire page image--see Figure 4-4." The intent here is to describe how a uniform pattern of ink may be made by abutting the pixel array defining the sampled black-and-white color area to itself in all directions. It does not mean that the pattern actually is applied to the page, but, rather, that it has been made large enough so that it would cover the entire page if it were applied through a mask as large as the entire page. Note also that the definition of the pixel array includes the transformation which brings it into the Device Coordinate System. This transformation establishes the position and orientation of the "tile" represented by the pixel array. The abutting process takes place relative to this initial position, which is not necessarily at the origin of the Device Coordinate System.

The Interpress language specification then goes on to say "The color deposited through the mask on an image pixel corresponding in position to a color sample value of x depends on x." The mask is a stencil through which ink is deposited. Only those positions in the image which correspond in position to the holes in the mask can be affected by the masking operation which applies ink to the image. In the case of sampled black-and-white ink which is created through the use of a pixel array two different imaging interpretations can be applied to the 0's in the array as a function of the parameter *transparent.* The 1's in the array alway apply black ink through the mask. The value of the parameter *transparent* determines what will happen to the image at points in the image which correspond to holes in the mask and 0's in the ink. If the parameter *transparent* has the value 0 white ink is applied, thereby obliterating any previous image pixel which might have been deposited at that point. If the parameter *transparent* has the value 1 no ink is applied, thereby leaving unaltered any previous image pixel which might have been deposited at that point.

4.8 Mask operators

Care must be taken not to confuse the process of creating a mask with that of applying ink through the mask thus created. The reader has a tendency to think of the process of creating a mask in terms of drawing lines on a paper. If he does this he creates a mental picture of an image being deposited on a page at the time the mask is created, and this is an incorrect view. A more appropriate mental image to retain concerning mask creation is that of the creation of a shape in the form of a stencil. If the stencil is created by trajectories and outlines the creation process can be thought of as wielding a pair of scissors on a piece of stencil material. If the stencil is created through the use of pixel arrays the creation process can be thought of as wielding a pixel sized punch on a piece of stencil material, where the punch is applied at each position occupied by a 1 in the pixel array. At some later point in time this stencil will be transformed in size and orientation, laid down at some location on the page image, and then ink will be deposited through it. It is this latter series of operations which Interpress refers to as *masking.*

Note that *masking* causes ink to be deposited through a mask. There are three kinds of masks which are referred to in this section. The first two are created by trajectories and outlines.

(See the definitions of *types* in Part I). The MASKFILL operator causes the interior of an outline to be treated as the stencil through which ink is deposited on the page image.

The MASKSTROKE operator causes a trajectory to be treated as the centerline of a stencil whose width about that centerline is controlled by an imager variable *strokeWidth*, and the shape of whose endpoints are controlled by an imager variable *strokeEnd*. In this latter operation think of the trajectory as being drawn on a piece of stencil material, and then apply a scissors to cut out the material along the trajectory at a distance *strokeWidth*/2 on either side of the trajectory. The imager variable *strokeEnd* controls how the endpoints of the hole cut in the stencil are to be shaped. Then apply ink through that stencil.

The third type of mask is created with a pixel array. In this case the pixel array is viewed as though a pixel sized punch has been used to punch a hole through a stencil material at each pixel position occupied by a 1 in the pixel array, and the stencil material has been left unaltered at each pixel position occupied by a 0 in the pixel array. The MASKPIXEL operator causes ink to be deposited through the holes in the stencil thus created.

In all of these masking operations the mask is defined in the creator's **MCS**. Hence, before the ink is applied in the printer's **DCS** the current transformation *T* must be applied to the mask to transform it into the printer's coordinate system. If the ink which is being applied is of the sampled black-and-white variety the current transformation *T* is also applied to the ink before it is deposited through the mask.

4.9 Character operators

This section is quite clear in the Interpress 82 specification, and should be read in its entirety before proceeding to Sections 4.9.1 and 4.9.2, below. A portion of Section 4.9 is included here, however, so that this section of the Reader's Guide can be read in conjunction with Sections 4.9.1, 4.9.2, and 4.4.6 as a coherent set of information in its own right.

The important concepts to understand about the printing of characters are the following. A character is imaged by executing a special type of composed operator called a *character operator.* Instances of characters are placed on a page by invoking these character operators in conjunction with suitable transformations. In order for a creator to be able to anticipate the effect of these invocations it is necessary for all character operators to obey a common set of imaging rules.

A character operator performs three operations:

1. *Generates masks.* It invokes mask operators to specify the mask or masks that define the shape of the character, thus causing an image of the character to be added to the page image. The placement, size, and orientation of the mask are controlled by the current transformation.
2. *Moves to the next character position.* It alters the current position so as to prepare for the next character in a sequence. It does so by adding a function of the "character width" to the current position. This function will be described later when we discuss the processes of *adjusting* and *correcting* text. For purposes of the following discussion consider that the completion of the imaging of a character results in the adjustment of the current position by an amount equal to the *width* of the character.

3. *Corrects spacing.* This operation is associated with the processes of *adjusting* and *correcting* text which are described later. For purposes of the following discussion consider that this operation results in no changes to the position of the character.

From the viewpoint of the imaging model of Interpress a *character mask* is a stencil whose shape is pre-defined. It is important to understand that the character operators of an Interpress font are representations of the shape of a character, and not of its imaging size. It is a convention of Interpress that a font is defined in a Standard Coordinate System which does not have physical dimensions associated with its coordinates. All character operators in the font would produce characters whose effective printing size would be one unit in height. Thus, in the absence of any scaling operation, all character operators in the font would produce characters whose effective printing size would be one meter, because they would be brought into the Interpress Coordinate System as unit high elements, and the unit of Interpress is one meter. The phrase "effective printing size" is used to convey the concept that a character would be imaged so that the origin of a similar character on the following printing line would be located one meter below the origin of this character under the condition that no additional "leading" exists between the lines. In conventional printer terminology the "point size" of the type corresponds to the phrase "effective printing size". Thus, the one unit high character includes the standard inter-line space which is built into each character.

At the time of the imaging of a character this *character mask* is generally subjected to a transformation which may rotate and/or reflect it to any possible orientations, and which may also scale it to any size. (It is important to understand that there is no concept of "*portrait*" or "*landscape*" inherent in the character operators of a font. In raster type printers characters are often defined by stored raster representations. These rasters must be created relative to the scanning sequence of the raster image generator in the printer. Thus, characters must be scanned differently relative to each orientation in which they may be imaged on a page. Each such scanning results in a different "font" from the printer's viewpoint even though they correspond to a single font from the Interpress viewpoint. Thus, the introduction of "portrait" and "landscape" character orientations are a necessity of the printer implementation, not of the font character operators *per se.*)

From a formal presentation viewpoint we should discuss the concepts of *adjustment* and *correction* at this point in this Reader's Guide. In the interests of continuity of presentation of other concepts at this point we shall defer doing so until the end of our discussion of Section 4.9.

4.9.1 Fonts

A font is a 5-element vector, whose first element is a vector of operators, one such operator for each character contained within the font. (The other components of the font vector are the *metrics, characterMetrics, name,* and *extras.*) A character operator is located in the vector of operators by means of its numeric character code which serves as the integer index into that vector. It is important to understand that each character is represented in the font by a *character operator.* Because it is an *operator* its execution performs a number of functions. These functions may change the size and orientation of the character which is placed on the page. See Section **4.9.2** for a description of these functions.

A font is identified by a hierarchical naming convention which is well described in Section 4.9.1 in the specification. For completeness of exposition here we will repeat the description of locating a font which a creator wishes to use in a document. The location of fonts which are to be used frequently throughout the document generally occurs in the preamble. The location of a font which is unique to a specific page may occur within the *PageBody* of the page itself. In either case the function is accomplished with the operator FINDFONT which takes as an operand a vector *v* of *identifiers* which are the hierarchical name of the font. The result of executing FINDFONT is to locate the font in the printer's library and place it on the stack. This location process is external to the Interpress operations, and is accomplished in a manner which is appropriate for each printer. If the printer has a font of the designated hierarchical name in its library this external process returns that font to the stack, and reenters the Interpress domain. If it does not have a font with the designated hierarchical name the external process may locate a font which it feels is an acceptable substitute for the designated font, return that substitute font to the stack, and reenter the Interpress domain. Other suitable actions might also take place in the absence of the font, such as aborting the job or the page, as a function of the printer's implementation. Appropriate warning or error messages are also returned if the font is not available in the printer's font library.

Remember that the font returned to the stack is a vector of operators which will generate one meter high characters in the absence of any scaling operations.

4.9.2 Modifying a character vector

It is generally the case that the original creator of a document has an intended font imaging size in mind at the time he composes a document. The MODIFYFONT operator provides him with the capability to provide a transformation *m* which, in the absence of any other transformation, will scale the one unit high characters of the font obtained by the FINDFONT operator to the size he desires for printing. The MODIFYFONT operator takes two operands from the stack. The top of the stack must be a vector of operators, and generally is a font. The second element on the stack must be a transformation, *m*. The result of executing the MODIFYFONT operator is to create a new vector of operators in which each operator has been replaced by a new operator whose execution calls for the modification of the old operator by a concatenation of the transformation *m* with the current transformation *T* which is in effect at the time this new character operator is invoked. The transformation *m* is generally a scaling transformation which scales the one unit high characters of a font located by a FINDFONT operator to the point size and orientation desired by the original document creator. Thus, the effect of MODIFYFONT is to create a new vector of operators which will produce character images of the size and orientation desired by the original creator of the document, *if the current transformation T only performs a scale change from the units of the **MCS** to the **DCS** with no rotation or magnification.* For example, if the original creator of the document desires a 10-point font oriented with its vertical axis parallel to the y-axis of his **MCS**, and he is using inches as his dimensional units the transformation *m* would be created by the sequence 10/72 SCALE. If he desired a 10-point font oriented with its vertical axis parallel to the x-axis of his **MCS**, and he is using inches as his dimensional units the transformation *m* would be created by the sequence 10/72 SCALE 90 ROTATE CONCAT. If he were using points as his dimensional units these transformations would be created by the sequences 10 SCALE, and 10 SCALE 90 ROTATE CONCAT, respectively. If the printer's **DCS** were expressed in units of 1/300th's inches and the current transformation had no rotational nor magnification effects the values of 10/72 inches or 10 points would be converted to their equivalent value in 1/300th's inches, namely 41.667 (i.e. 10/72 inches = 10 points = 41.667/300th's inches).

It is important at this point to distinguish between the original creator of a document, and a potential later user of that same document. An example will illustrate this point. The original creator of a document may have in mind the printing of the document on 8-1/2 by 11 paper in a portrait mode. Assume that for the purpose of creating the images of this document he uses an **MCS** with origin at the lower left hand corner of a page with his y-axis along the long edge of the paper. Assume that he wishes to use a 10-point Helvetica font. (This size has been chosen for this illustration to be compatible with the description of the SHOW operator in Section 4.4.6, which see.) A later user of *this same document* may wish to print it two-up on 8-1/2 by 11 paper in a landscape mode. We shall follow the actions of the original creator of the document, and return to this later user in a while.

The document creator will repeat the font locating process and modifying process for each font, and each font size and orientation, which he requires. Each such operation produces a different font. In general these operations will take place in the preamble, and the fonts thus created will be stored in the *pageFrame*. The storage process is accomplished by the sequence *n* FSET which stores the top of the stack in the *pageFrame* vector at the location indexed by *n*. Thus each font is stored at a location *n* known to the document creator. Any font may be retrieved by the sequence *n* FGET which returns it to the top of the stack.

When the creator wishes to use a font for imaging a particular set of text he does so by setting the font into the Imager Pool variable named *showVec* which has an index value of 12. He does so by means of the convenience operator SETFONT which takes an integer *n* from the top of the stack, uses *n* as an index into the *pageFrame* to obtain the font which has been stored there in the manner described above, and stores the font so obtained into the Imager Pool variable named *showVec*. Until the contents of *showVec* are changed by another SETFONT it is this font which will be used when subsequent SHOW operators are invoked. (See Section 4.4.6.)

It would appear from the above description that once a particular font (e.g. the 10-point font described above) is established in the Imager Pool variable named *showVec* characters of that size and having an orientation corresponding to his **MCS** coordinate system would be imaged on the page when the SHOW operator is invoked. It must be remembered, however, that each element of a font is an *imaging operator.* Because such an element is an *operator* it may perform a number of functions before (and after) it actually lays down the image of its associated character. Particular to the current point is the fact that *each character operator in the font concatenates its associated transformation m with the current transformation T so that the transformation m*T is the effective transformation at the time of the imaging operation.* A clear understanding of the italicized phrase above is critical to the understanding of the character imaging process. The *current transformation, T,* which is active at the time the SHOW operator is invoked is applied to each character operator during the processing of the SHOW operator. Thus, although the Imager Pool variable *showVec* might contain a font with a transformation *m* which will produce a 10-point font having a given orientation, the current transformation *T* when concatenated with *m* to form *m*T* might produce a font having an altogether different size and orientation during the imaging process. This conversion does not change the contents of the Imager Pool variable *showVec*. It only changes the images which are produced by the operators contained in the font which is installed in *showVec.*

The original creator of the document will have seen to it that the current transformation *T* is set to the proper state to obtain his 8-1/2 by 11 inch portrait page with the 10-point font held in the Imager Pool variable *showVec* printing as a 10-point portrait font. Now, let us return to the later user of that same document who wishes to print it two-up on 8-1/2 by 11 paper in a landscape mode. He merely needs to establish a current transformation *T* which rotates the image 90 degrees, scales it by .6, translates it to a new origin, and then *invoke the original creator's Interpress master without any further change to its internal content*, in order to print that same page in his desired two-up orientation. His current transformaton will be applied at the time of the execution of the SHOW operator so that the character operators of the 10-point font installed in the Imager Pool variable *showVec* will have been changed to produce 6-point characters rotated by 90 degrees, with the entire image shifted to a new location on the page.

4.9 (Revisited) Line justification

Section 4.9 is being picked up where it was bypassed for the sake of continuity.

Justification of text (and other images, if desired) on a line between fixed margins is provided within Interpress. Associated with each character operator is a pair of parameters, *widthX* and *widthY*. In general, for western languages *widthX* is positive and *widthY* is 0. Similarly, for languages read right-to-left *widthX* is negative and *widthY* is 0. For languages read top-to-bottom *widthX* is 0 and *widthY* is generally negative. If justification is not being used the position of the next character to be printed on a line is established by the sequence *widthX widthY* SETXYREL which is automatically invoked within the execution of the character operator itself. While the general rules for width definition are as described above, Interpress 82 does not constrain the font designer to adhere to any specific standard with respect to his designation of *widthX* and *widthY*. He is free to use these parameters in any way he may choose so long as he recognizes that they will be used in the manner characterized by *widthX widthY* SETXYREL.

A parameter *amplifySpace* in the Imager State provides a means for achieving justification of text between fixed margins. For each font there is a standard *"space" character* which deposits no ink, and whose width is used to separate words. There may be more than one character in the font that has the property of the standard "space" character in the sense that such a character deposits no ink on the paper, but is used to provide spacing between printed objects. A character with this property is called a *spaceband* character. Some spaceband characters may be required to retain their width when a line is justified, others may not be required to do so. An example of a spaceband character that may be required to retain its width is a special space character incorporated in the printing of a mathematical formula. If the entire formula had to be slightly adjusted to fit a given space such a character would be adjusted in position so as to maintain the proper proportions of the whole formula. However, this type of space character would not be permitted to grow independent of the rest of the formula in order to justify a line containing the formula. Those spaceband characters which permit their width to be increased are called *amplifying characters.* Note that there may be "space" characters in a font which have the same properties as printing characters. The *"figure space" character* whose width is set to match the standard width of the digits 0 to 9 is an example of a such a character.

Justification of a line of text between fixed margins is achieved by increasing the space between words. This is accomplished by summing the widths of the characters on the line (including the spaceband characters) and determining the difference between that sum and the desired length of the line. For the following discussion call this difference the *error.* We assume that the number of characters to be included on a line is chosen so as to make the value of the error positive. The error can be removed by modifying the effective width of the *amplifying* characters. This is accomplished in the following fashion. We wish the error to be taken out by changing the amplifySpace factor. Therefore, we set:

error = amplifySpace * (Sum-of-widths-of-amplifying-characters),

where,

Sum-of-widths-of-amplifying-characters =
SQRT((Sum of WidthX)**2+(Sum of WidthY)**2))
where the sums cover all amplifying characters on the line.

We then solve the first equation for amplifySpace, i.e.:

amplifySpace = error/(Sum-of-widths-of-amplifying-characters).

When the line is printed justification is achieved by multiplying the width of each *amplifying* character by the factor *amplifySpace.* This is accomplished by having each *amplifying* character establish the position of the next character to be printed on the line by the sequence *widthX*amplifySpace widthY*amplifySpace* SETXYREL which is automatically invoked within the execution of the *amplifying* character operator itself.

4.9.3 Metrics

Almost all of the concepts in the section on character and font metrics are straightforward and easy to follow. The one exception to this is the font metric named *easy.* The *easy vector* provides information on the sizes and orientations of fonts that the printer can readily produce. Interpress provides the creator with a capability to call for fonts of arbitrary size, and having any orientation. In raster type printers characters are often defined by stored raster representations. Such printers generally do not have raster representations for every size and orientation of a font. The sizes and orientations that it does have are the *easy* ones. If it does not have any raster representation in the orientation called for by the master the text in that orientation cannot be printed. Such text will not be printed and an appearance error will be reported. If the printer has a raster representation of some other font with the right orientation, it will make a font substitution according to an appropriate algorithm, and print the page with an appearance warning.

From the Interpress viewpoint a font is a collection of operators each of which contains, among other things, a mask contained within a unit high area. This mask is scaled and rotated at imaging time so that it is of the proper size and orientation to create the desired imaged. From a practical viewpoint an Interpress font explodes into a collection of printer fonts at a printer. Each such printer font contains a set of character operators each of which produces an image with a pre-defined character size and orientation. The *easy* vector defines the members of the set of printer fonts.

The elements of the *easy* vector may be defined by the transformations which are used to modify the single Interpress font to the multiple printer fonts which are available at the printer. The profusion of units which different creators may employ creates a problem for the expression of the elements of the *easy* vector. Interpress resolves this problem by expressing the defining transformations of the *easy* vector in the units of the Interpress Coordinate System.

4.10 Space correction

The methods used to make fine adjustments of spacing within a line of text at the printer is formally described in the Interpress specification by means of Pascal-like programs. This section of the Reader's Guide will provide an alternative description of these methods couched in conventional prose. The intent here is to be comprehensive and self-contained. Hence, this section will repeat portions of the Interpress language specification. The reader is again reminded that the Interpress language specification is the authoritative source for this information. Any conflict between the reader's interpretation of this description of these methods and the one presented in the Interpress language specification must be resolved by reference to the latter document.

Sometimes the exact positioning of a mask must be computed when the master is printed rather than when it is created. This is often the case if positioning depends in detail on the widths of characters, because the imager may not be able to use a character font that has widths that are identical to those available when the master was generated. Of course, if the creator knows the properties of the font exactly, no new computation by the imager will be necessary. The creator will make a master that specifies the exact position of each mask.

Interpress provides a mechanism to *correct* a set of masks, which is used most frequently to insure that a line of characters intended to appear uniformly justified between margins are in fact so justified. Correction is achieved by expanding or contracting some "correction space" until the characters fit in the desired space. The Interpress mechanism is not specific to characters, but will correct any kind of mask.

It should be noted that the correct process is essentially a "re-adjust" process. It is not intended that the correct operation be used as a substitute for, nor alternative to, the justification operation. The *amplifySpace* mechanism provides the primary justification operation.

Mask correction is achieved with the CORRECT operator, which takes as its only argument a body containing the operators that invoke all of the masks that are to be corrected. Generally this body contains operators which will generate a line of text, but it need not be restricted to that. The desired length of the line is established through the use of the SETCORRECTMEASURE operator. SETCORRECTMEASURE establishes the values of *correctMX* and *correctMY* in the imager state. The line to be corrected has a length equal to the square root of the sum of the squares of *correctMX* and *correctMY*. CORRECT will generally execute the body *twice*, first to compute how much correction is required, and then a second time to actually create the image. When CORRECT is initiated the current position is recorded, and the body is executed, but the character operators are not allowed to modify the page image (the variable *noImage* is set to *1*). The execution of the body invokes each character operator. Each character operator, in turn, invokes either the CORRECTSPACE or CORRECTMASK operator. If the character's width can be adjusted to remedy spacing problems the operator calls *widthX widthY* CORRECTSPACE if the character is not amplifying, or *widthX*amplifySpace widthY*amplifySpace* CORRECTSPACE if it is amplifying. If the character's width should not be adjusted (e.g., a character which deposits ink, or a "figure space" designed to equal precisely the widths of the figures 0..9), the operator calls CORRECTMASK.

During this first pass amplifying characters are amplified by the multiplier set in the *amplifySpace* parameter in the imager state. The result of this first pass is the determination of four parameters:

1. The endpoint of the line that will result if the line is printed under the current conditions.
2. The difference between this endpoint and the desired endpoint established by SETCORRECTMEASURE.
3. The number of space characters which may be modified to achieve the required re-adjustment.
4. The number of "non-space" masks which may be re-adjusted if the line is too long and all of the required space cannot be taken out of the available space characters.

Three conditions may exist at the end of the first pass. The error in the length of the line may be acceptable. There is a default parameter which is designated by the name *correctTolerance*, and equal to the value of the square root of the sum of the squares of *correctTX* and *correctTY* in the imager state. If the difference between the endpoint of the line that will result if the line is printed under the current conditions and the desired endpoint established by SETCORRECTMEASURE is less than the *correctTolerance* no further correction is required. If not a special correction procedure is invoked if the creator has called for a CORRECT operation.

If the line is too short the CORRECT operation will cause the space characters to be increased in size in proportion to their original sizes to the point where the line fits the desired space. Note that their original sizes include the *amplifySpace* factor in the case of amplifying characters.

If the line is too long the CORRECT operation will cause the space characters to be reduced in size in proportion to their original sizes to the point where the line fits the desired space. However, there is a threshold below which spaces will not be reduced. A parameter *correctShrink* in the imager state establishes the minimum value that a space can be reduced to by the CORRECT operator. A space is never allowed to be reduced to less than (1 - *correctShrink*) times its former size. The sum of all the widths of the space characters multiplied by the parameter *correctShrink* determines the maximum amount of space that may be removed from the line by shrinking space characters. If the amount of space to be removed from a line exceeds that value the line cannot be corrected merely by reducing the size of the space characters. In this case the space between words is reduced to the minimum allowed value as computed from *correctShrink*. The remainder of the space left to be removed from the line is compared with the *correctTolerance*. If it is less than the *correctTolerance* no further action is required. If it is greater than the *correctTolerance* the remainder of the space is removed by reducing the space between all non-space characters by an equal amount. This amount is determined by dividing the remaining space to be eliminated by one less than the number of non-space characters. (The reason for the reduction by one is that there are n-1 opportunities to decrease the space between n characters.)

Presented in a somewhat informal way, the entire justification and correction process is performed in the following fashion, assuming that the creator wants to invoke the full CORRECT operation:

1. The creator justifies the line in his domain using his knowledge of the font widths. To do so he calculates where the line will end if he breaks it off after each word. When he reaches the threshold where he can no longer contain the next word he makes his line ending decision. At this point he calculates how much space he must add to the line, and determines the *amplifySpace* factor which must be applied to the amplifying characters contained within the line.

2. When the line is to be printed the amplifySpace factor is set in the imager state. This is accomplished by getting the value to be placed in the amplifySpace factor onto the stack and following it with the sequence 18 iset. (18 is the index value of the amplifySpace factor in the Imager State vector.)
3. The desired length of the line is set in the imager state using a setcorrectmeasure operator.
4. The correctTolerance is set in the imager state using a setcorrecttolerance operator.
5. The correctShrink is set in the imager state using the sequence 20 iset. Presumably this is done once at the beginning of each page as part of the set-up of the imager state. (See the section on Imager Variables where it describes how this process is implemented by means of a composed operator created by the preamble.)
6. Line printing is invoked with a correct operator.
7. The correct operator sets noImage to the value 1, and correctPass to the value 1 in the imager state, and invokes the show operator.
8. The show operator, with noImage in the 1 state, invokes the character operators. Each character operator, in turn, invokes either the correctspace or correctmask operator depending on whether or not it is a spaceband character. On this pass each spaceband character causes correctspace to be called with its width multiplied by the amplifySpace factor if it is an amplifying character. If it is a non-amplifying character correctspace is called without the multiplication by the amplifySpace factor. Each non-space character calls correctmask, and has its width determined from the font widths table in the printer.
9. At the end of the first pass CORRECT knows the amount of difference between the position where the line will end if it uses the *amplifySpace* factor and font width factors, and the desired end position of the line established by the SETCORRECTMEASURE operator. If this difference is less than the value of *correctTolerance* the process proceeds to image the line as it is with no further correcting operations. If not, the following steps are processed. At the end of the first pass CORRECT also knows the number of space characters in the line, and the number of non-space characters in the line.
10. CORRECT now calculates how it will correct out the difference determined in step 9 during the second pass over the line. If the line is too short the spaces will be increased in length to fill out the line, and the process skips to step 12. If the line will be too long, the process proceeds to step 11.
11. A factor f is now computed. f is defined to be the quotient obtained by dividing the amount of space to be taken out of the line by the sum of the spaces occupied by space characters. If f is less than or equal to correctShrink the line can be corrected merely by reducing the size of the space characters. In this case the space characters will be reduced by the factor *f*, and the line will be properly corrected without violating the space reduction threshold set by *correctShrink*.

    If *f* is greater than *correctShrink* the line cannot be corrected merely by reducing the size of the space characters. In this case the space characters will be reduced to their minimum allowed value by multiplying all of them by the factor *correctShrink*. The remainder of the space left to be removed from the line is compared with *correctTolerance*. If it is less than *correctTolerance* no further action is required. If it is greater than *correctTolerance* the remainder of the space will be removed by reducing the space between all non-space characters by an equal amount. This amount is determined by dividing the remaining space to be eliminated by one less than the number of non-space characters. (The reason for the reduction by one is that there are n-1 opportunities to decrease the space between n characters.) During the second pass of the CORRECT operator each space character will be reduced to its minimum value and each non-space character's width will be reduced by the amount just calculated. since the calculation will generally not produce an integer for each character space reduction, the reduction process must be properly implemented to avoid round-off errors.

12. The CORRECT process now proceeds to pass 2. It sets the *noImage* parameter to *0*, and the *correctPass* parameter to *2* in the imager state, and again invokes the SHOW operator. During this pass the mask position corrections and space character length determinations made by the previous steps are invoked to bring the line to its desired length.

Appendix A References

Xerox Corporation. *Interpress 82 Electronic Printing Standard.* Xerox System Integration Standard. Stamford, Connecticut; January 1982; XSIS 048201.

This document provides a precise definition of the Intepress 82 language that is elaborated upon in this Interpress 82 Reader's Guide.

7-Bit Coded Character Set for Information Processing Interchange. ISO 646-1973 (E). International Standards Organization.

This document defines a limited character set for infor information interchange. It is almost compatible with ASCII. The Interpress 82 uses of ISO 646 are restricted to the subset that is compatible with ASCII.

Xerox Corporation. *Xerox Character Code Standard.* Xerox System Integration Standard. Stamford, Connecticut; To be published.

This document defines the 16-bit codes assigned to the characters and symbols supported by Xerox products.

Xerox Corporation. *Xerox Character Encoding Standard.* Xerox System Integration Standard. Stamford, Connecticut; To be published as an Appendix to the *Xerox Character Code Standard.*

This is an Appendix to the *Xerox Character Code Standard.* It defines a character encoding system which compresses the amount of information required to define a sequence of characters. The compression technique also results in the transmission of codes in ISO 646 in an unaltered form.

B

Appendix B
Support levels and mapping

B.1 Interpress support levels

Different printers support differing combinations of functions handled by the Interpress language, depending on their individual limitations and capabilities. All printers described as supporting Interpress must be capable of implementing a basic subset of functions, comprising the following:

Text -- Basic font set capabilities.

Vector Graphics -- Trajectory-defined images, limited to rectilinear forms aligned with the edges of the printed page.

Ink -- Black.

Language -- The Interpress 82 language structure and its transmission encoding as described within the specification section of this manual, with the exception of the IF operators and computational operators.

Storage -- Data element size constraints as described in the specification section.

Within each of the above functions, levels are designated to identify the extent of support that a printer is capable of achieving. For example, within the Ink function, a printer that supports only black ink values is at ink level zero, while a printer capable of producing gray values is at ink level one.

B.2 Mapping

As illustrated below, all element level designations are combined to form a map for the specified printer, following the form:

Language.Pixel Arrays.Ink.Vector Graphics.Text.Storage.Clipping

A printer does not always support the Interpress elements at equal levels; typically it supports some Interpress elements at level zero (for example, black ink but not gray) and other elements at level one (for example, pixel arrays). Such a printer is considered to belong to the level zero product class, and not to level one. For implementation purposes, however, the Interpress Map is specific to the product. This allows the user of an Interpress master to predict the default actions (if any) of the printer being called upon to implement the Interpress master.

	Language	Pixel Arrays	Ink	Vector Graphics	Text	Storage	Clipping
	81 Interpress 81 82 Interpress 82	0 No 1 Scan-sequence aligned	0 Black 1 Gray	0 Lines parallel to device axes 1 Open or filled polygons	0 Easy fonts 1 (not available)	0 Size constraints 1 (not defined)	0 No 1 (not available)
Basic Support Level	82	0	0	0	0	0	0
9700	82	1	0	0	0	0	0
8044	82	0	1	1	0	0	0
5700	82	1	1	1	0	0	0

Figure B.1 Interpress Mapping Concept

Appendix C
Compression specifications

C.1 Compressed pixel vectors

The following paragraphs address the format of compressed binary raster images that are to be decompressed for printing on PSD Interpress printers. The compressed image is "contained" in the sequenceCompressedPixelVector transmission token and any immediately following sequenceContinued tokens, each of which consists of descriptor fields followed by the compressed data. (See "Token Formats" in Part I of the Reader's Guide.)

Ultimately Xerox may support many different compression algorithms. Currently it supports the single algorithm described in this Appendix. This algorithm has been selected for a number of reasons. The two primary ones are the following:

1. The algorithm provides an acceptable level of compression for raster images of the class that are most likely to be used within printings systems having a resolution in the neighborhood of 300 black-and-white pixels per inch.
2. The algorithm can be efficiently implemented on the main frames of the major computer manufacturers.

C.2 Compression algorithm principles

The compression algorithm uses a combination of a prediction mechanism and a run length encoding mechanism. Because of the second point above, the prediction mechanism is extremely simple, consisting merely of a single bit "Exclusive Or". The bit selected for this process can either be the corresponding bit on the preceding scan line, or a bit on the current scan line located 5, 6, 7, or 8 bit positions preceding the bit being predicted. (The rationale behind this latter set of selections is to provide a reasonable mechanism within the same "Exclusive Or" framework to deal with half-tone cells of size 5x5, 6x6, 7x7, and 8x8, the assumption being that there is likely to be a high correlation between corresponding bit positions in adjacent cells.) The bit selected for the predictor in this latter case is set at the beginning of the compression operation for the entire image. Thus, the compression algorithm for an image contains two different predictor possibilities, the second one of which is selected from among four possibilities at the beginning of the compression operation. The predictor used is unique to each scan line, and can be arbitrarily varied from scan line to scan line.

Compression modes

Each scan line begins with a *Line Control Code* (LCC) that designates which of the following four compression modes is to be used for the scan line. The four compression modes are:

1. RAW, in which case the original raster information is presented without modification, i.e. no prediction and no run length encoding.
2. ENC, in which case no predictor is applied to the original raster information, but it is run length encoded.
3. HTN, in which case the predictor is the bit preceding the current bit on the same scan line by 5, 6, 7, or 8 bits. (Which of these cases to apply is set by a parameter at the beginning of the compression operation.)
4. LIN, in which case the predictor is the corresponding bit on the preceding scan line.

The run length encoding technique is a Huffman code of Xerox's design. It is based on the use of a 4-bit input data group which is referred to as a *nibble.* Therefore, the original input data stream must be a multiple of four bits. In fact, for other reasons, the original input data stream must be a multiple of eight bits. The run length encoding system also produces a compressed data output stream that is a multiple of eight bits.

Nibble classification

The run length encoding process breaks the data to be encoded into nibbles. These nibbles are then classified into four categories as follows:

1. Z, a nibble containing the value 0000 (four zeros).
2. A, a nibble containing any of the values 0001, 0010, 0100, or 1000 (i.e. precisely one 1).
3. B, a nibble containing any of the values 0011, 0101, 0110, 0111, 1001, 1010, 1011, 1100, 1101, 1110, or 1111 (i.e. any of the values not included in Z or A).

The data to be run length encoded is then broken into sets of nibbles, each set of which consists of a sequence of *n* Z type nibbles ($n = 0, 1, 2, \ldots$) terminated by an A or B type nibble. The run length encoding process then uses a set of special codes (of lengths 4-, 8-, 12-, 16-, 20-, and 24-bits) to designate the number of Z type nibbles in each set, as well as the A or B type nibble which terminates the set. Note that the number of Z type nibbles in a set may be zero.

Item size

In the definition of a compressed image we will use the following nomenclature with the following designated meaning:

1. Word: 2 consecutive bytes. Transmitted most significant byte first. Unless otherwise specified, a word is not required to begin on any particular (word/byte/nibble) boundary.
2. Byte: 8 consecutive bits.
3. Nibble: 4 consecutive bits.

Definition format

The definition of a Compressed Image is given in a format similar to the Backus-Naur convention. In this format each "paragraph" of text in the definition characterizes an element of the compressed image. For example, the first element of the compressed image is a word containing 16 binary zeros. This is defined by the statement "Word: Reserved, must be binary zero". The second element of the compressed image is a word containing the value NRange. The value NRange is constrained to lie between 5 and 8, inclusive. This is defined by the statement "Word: NRange [5..8]".

Specific literal values are designated by the enclosing them in "/" symbols. For example, the specific literal value 01110001 is designated as /01110001/. Literals which are defined as the selection of one among a set of mutually exclusive alternatives are designated by separating the alternatives with the symbol "|". For example, the literal which is the selection of one among the set of mutually exclusive alternatives 0001, 0010, 0100, and 1000 is expressed as /0001|0010|0100|1000/.

C.3 Compression algorithm specification

With this background the full compression algorithm can be stated in the following semi-formal manner:

Compressed Image

Word: Reserved, must be binary zero

Word: NRange [5..8]

Word: Scan line length. Scan line length is an integer which defines the number of pixels in a scan line. This vaue must be a multiple of eight.

Byte: SOI (Start of Image) /01110000/

Lines: one or more compressed Data Lines. Data lines are nibble oriented. Each Data Line is composed of a Line Control code (LCC) followed by a line of data. This data may end on a nibble boundary. Data Line is defined below.

Byte: EOI (End of Image) /01110001/

Nibble: Pad /0000/ One or more Pad nibbles are required only when the EOI terminates on a nibble but not on a 16-bit word boundary.

Data Line

Byte: LCC (Line Control Code) one of four codes which control processing of the remainder of the data line. The remainder of the data line is termed a *Run* (See definition of a Run, below). The four LCC codes are RAW: /00000000/, ENC: /0000010/, LIN: /00000001/, or HTN: /00000011/.

RAW specifies that the bits of the Run which follow are pixels. The number of bits is specified by the value Scan line length, above. Scan line length must be a multiple of four.

ENC specifies that the bits of the Run which follow are pixels which have been compressed. The compression process is accomplished purely by run length encoding. No prediction algorithm has been employed.

HTN specifies that the bits of the Run which follow are compressed. The compression process is accomplished by a combination of a prediction algorithm and a run length encoding algorithm. In this mode each bit has been exclusively ored (XORed) with the bit that is N bits back in the current line. The value of N is defined by NRange above. The results of this exclusive oring process are then run length encoded. Leading zeros are assumed for the initial bits necessary to compare the first NRange bits of the line.

The first Data line and each 16th line thereafter of an image must be RAW, HTN or ENC.

LIN specifies that the bits of the Run which follow are compressed. The compression process is accomplished by a combination of a prediction algorithm and a run length encoding algorithm. In this mode each bit has been exclusively ored (XORed) with the bit that is in the corresponding position in the preceding scan line. The results of this exclusive oring process are then run length encoded.

Table C.1 Line types v. encoding functions

LCC	XORed	Run Length Encoded
RAW	no	no
ENC	no	yes
LIN	yes	yes
HTN	yes	yes

Raster line

The Run is either raw pixels ('RAW' above) or compressed according to the following method:

First, we define the following:

Z: nibble = /0000/

A: nibble = /0001|0010|0100|1000/ (A nibble with a single 1 bit.)

B: nibble = /0011|0101|0110|0111|1001|1010|1011|1100|1101|1110|1111/
(A nibble with more than a single 1 bit.)

(Z, A, B are the mutually exclusive and collectively exhaustive cases of 4 bits)

T: nibble = /A|B/

H: The number of consecutive Z's in a sequence of Z's terminated by a T.

Then we define the encoding of a Run by the following Huffman encoding procedure:

Run: A series of one or more H,T pairs encoded as 4-, 8-, 12-, 16-, 20-, or 24-bit codes as follows:

Four-; eight-; and twelve-bit codes

Table C.2 4-bit codes 10xx:

H = 0, T = A where:

xx	A
00	1000
01	0100
10	0010
11	0001

Table C.3.1 8-bit codes 011xxxxr:

H = r = [0..1], T = B (0 or 1 Z nibbles, followed by a B nibble) where:

xxxx	B
0100	0011 (note xxxx not equal to B)
0101	0101
0110	0110
0111	0111
1001	1001
1010	1010
1011	1011
1100	1100
1101	1101
1110	1110
1111	1111

Table C.3.2 8-bit codes 0yyyyyxx:

H=yyy=[1..25], T=A (1 to 25 Z nibbles, followed by an A nibble) where:

xx	A
00	1000
01	0100
10	0010
11	0001

Table C.4 12-bit codes 11rrrrrrxxxx:

T=A=xxxx, H=rrrrrr=[26..63] (26 to 63 Z nibbles, followed by an A nibble)
T=B=xxxx, H=rrrrrr=[2..63] (2 to 63 Z nibbles, followed by a B nibble)

Sixteen-; twenty-; and twenty-four bit codes

16-, 20-, and 24-bit codes support the values 63<H<4096. They are created by a 12-bit code (11qqqqqq0000) followed by one of the preceding 4-, 8-, or 12-bit codes.

Table C.5 16-, 20-, and 24-bit codes

H=yyy=[1..25], T=A (1 to 25 Z nibbles, followed by an A nibble) where:

length	format	definition
16	11qqqqqq0000 10xx	H= qqqqqq*64, T=A(xx) (see 4-Bit Codes above)
20	11qqqqqq0000 011xxxxr	H= qqqqqq*64+r, T=B(xxxx) (see 8-Bit Codes above)
24	11qqqqqq0000 11rrrrrrxxxx:	H= qqqqqq*64+rrrrrr, T(xxxx)= (see 12-Bit Codes above)

End of line situation

Before encoding, a given line could have a stream of Z's without a T to terminate it at the end of the line. Under this condition, these 0's are skipped and an LCC for the next line is appended immediately to the encoded data stream. This LCC tells the decoder that all nibbles from the current position to the end of line are all Z's.

D

Appendix D A tutorial on the matrices used in Interpress

D.1 Introduction

This appendix has two purposes. First, it presents the matrices and matrix methods which are used within Interpress 82. Second, it demonstrates how to perform the mechanical operations involved in manipulating these matrices, and provides explicit application examples.

D.2 Interpress use of coordinate systems and transformations

One of the fundamental requirements of Interpress is that it permits a master to describe precisely the location of every image on the output page. Any such description requires that a coordinate system be established within each printer. Since the optimum coordinate system for each different type of printer may be different it is impossible to create a master in terms of printer coordinates, and still have the master be printer-independent. The solution to this difficulty is to introduce the Interpress Coordinate System (ICS), and express the precise location of all images on an output page in this coordinate system. Each printer can then provide a transformation which will convert Interpress Coordinates to its own Device Coordinates.

The creator of every master could be forced to express all of his coordinates in the ICS. Such a requirement would introduce a difficult and unnecessary constraint. Interpress 82 removes such a constraint, and permits the creator of a master to use whatever coordinate system he may choose. Further, it permits him to change his coordinate system dynamically during the creation of the images for a page.

The use of transformations permits the conversion of coordinates from any current Master Coordinate System (MCS) to the ICS, and thence to the Device Coordinate System (DCS). The use of vectors to represent points and matrices to represent transformations, together with the rules for matrix algebra, provides a simple and efficient method for implementing this transformation process.

Two critical operating principles govern the use of transformations within Interpress 82. *The transformation parameters are stated in terms of the **old** coordinate system.* Thus, for example, if the master wishes to create a new coordinate system whose axes are rotated 30 degrees counterclockwise from those of the old coordinate system it will contain a statement which says "rotate the coordinate system +30 degrees". *The transformation matrix which is created to represent the transformation is the one which will carry coordinates expressed in the* ***new*** *coordinate system into coordinates expressed in the* ***old*** *coordinate system.* Thus, in the example stated above, the transformation will generate the matrix which will convert coordinates expressed in the rotated coordinate system into those of the unrotated coordinate system.

Transformations are created in a sequence that moves backward from the DCS. That is, the system starts with the DCS, applies the transformation that shifts to the ICS, then applies the transformation that shifts to the first MCS (designated as BCS in this Reader's Guide), and so on. However, all transformations are directed toward the process of producing results in the Device Coordinate System (DCS). That is, each transformation defines how to convert coordinates from the new coordinate system back to its predecessor coordinate system, i.e. the one that is closer to the DCS. Thus, for example, the transformation that shifts from the ICS to the DCS is one that will convert coordinates expressed in the ICS to coordinates expressed in the DCS. When the master then wishes to shift to the BCS coordinate system the transformation that is given is the one that will convert coordinates expressed in the BCS into coordinates expressed in the ICS. Thus, the statements which create the sequence of transformations move away from the DCS, first to the ICS, then to the MCS. However, the sequence of transformations which these statements create always moves towards the DCS, progressing from the BCS to the ICS, and thence from the ICS to the DCS.

(Note: Interpress 82 designates all coordinate systems used in the master by the generic name Master Coordinate System (MCS). In this Reader's Guide we have designated the name Base Coordinate System (BCS) for the first such coordinate system introduced in the master. Subsequent coordinate systems introduced in the master are referred to as Local Coordinate Systems (LCS). The BCS and all LCS are merely specific instances of the MCS which can dynamically change during the creation of a page image.)

D.3 General matrix operations

We define a vector representing a point in a two-dimensional coordinate system to consist of three components whose values are x, y, and 1. We represent such a vector by a triad written as [x y 1]. Let us be specific at this point and introduce the vector representing a point in the DCS. Using our notation this vector will be written as $[x_D \ \ y_D \ \ 1]$, where the subscript D stands for Device Coordinates.

We define a matrix of the type used in Interpress to consist of 9 elements whose values are a, b, c, d, e, f, 0, 0, and 1. Such a matrix is represented by a three by three array written as:

$$\begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

We will show how such a matrix is used to represent a transformation. We will often identify matrices by a name such as T or **ID** (written in boldface), in which case it is understood that the name is merely a shorthand way of describing the associated three by three array. Thus, we may have a particular transformation, T_I, represented by a specific set of values for a, b, c, d, e, and f, and write it:

$$T_I = \begin{matrix} a_I & d_I & 0 \\ b_I & e_I & 0 \\ c_I & f_I & 1 \end{matrix}$$

and subsequently refer to the matrix by its name, $\mathbf{T}_I$.

Again let us be specific. We define the vector representing a point in the **ICS** by the notation $[x_I \;\; y_I \;\; 1]$, and define the matrix representing the transformation which converts Interpress Coordinates to Device Coordinates. We will name this matrix **ID**.

$$\mathbf{ID} = \begin{matrix} a_{ID} & d_{ID} & 0 \\ b_{ID} & e_{ID} & 0 \\ c_{ID} & f_{ID} & 1 \end{matrix}$$

We speak of transforming the point in **ICS** defined by the vector $[x_I \;\; y_I \;\; 1]$ to a corresponding point in **DCS** $[x_D \;\; y_D \;\; 1]$ by applying the transformation **ID**, described by its associated matrix. In notational form this is written as

$$[x_D \;\; y_D \;\; 1] = [x_I \;\; y_I \;\; 1]\,\mathbf{ID}$$

The operation described above is *defined* to be the *product* of the vector and the matrix. This product is defined as follows:

$$[x_D \;\; y_D \;\; 1] = [x_I \;\; y_I \;\; 1] \quad \begin{matrix} a_{ID} & d_{ID} & 0 \\ b_{ID} & e_{ID} & 0 \\ c_{ID} & f_{ID} & 1 \end{matrix}$$

where

$$x_D = x_I{*}a_{ID} + y_I{*}b_{ID} + c_{ID}$$
$$y_D = x_I{*}d_{ID} + y_I{*}e_{ID} + f_{ID}$$

The equations, above, expressing the relationship between x_D, y_D, x_I, y_I, and the elements of the matrix **ID** define the *product* of the vector and the matrix. Note that this "product" is not the conventional product associated with the multiplication of two numbers. It is a new kind of product which specifically defines the *product* of the vector and the matrix.

The equations for x_D and y_D are the most general linear equations which can be written. The introduction of homogeneous coordinates, with the third coordinate having the value 1, permits the vector by matrix multiplication to express this most general linear transformation in a compact and homogeneous manner. It also permits us to combine series of transformations in an efficient and easy to implement manner. We will now develop that process.

Now, let us suppose that we introduce a second transformation, **BI**, which transforms the point in the BCS defined by the vector $[x_B \; y_B \; 1]$ to a corresponding point, $[x_I \; y_I \; 1]$, in ICS by applying the transformation **BI**, described by its associated matrix. In notational form this is written as

$$[x_I \; y_I \; 1] = [x_B \; y_B \; 1] \, \mathbf{BI}$$

If we substitute our definition for $[x_I \; y_I \; 1]$ in terms of $[x_B \; y_B \; 1]$ and **BI** into the expression for $[x_D \; y_D \; 1]$ in terms of $[x_I \; y_I \; 1]$ and **ID** we have:

$$[x_D \; y_D \; 1] = [x_I \; y_I \; 1] \, \mathbf{ID} = [x_B \; y_B \; 1] \, \mathbf{BI \; ID}$$

Note that the **BI** matrix appears on the left in the product **BI ID**. This is important because the product **BI ID** is *not equal* to the product **ID BI**. Mathematically speaking, matrix products are not commutative.

Following our definition of an Interpress matrix, we have:

$$\mathbf{BI} = \begin{matrix} a_{BI} & d_{BI} & 0 \\ b_{BI} & e_{BI} & 0 \\ c_{BI} & f_{BI} & 1 \end{matrix}$$

and

$$[x_I \; y_I \; 1] = [x_B \; y_B \; 1] \quad \begin{matrix} a_{BI} & d_{BI} & 0 \\ b_{BI} & e_{BI} & 0 \\ c_{BI} & f_{BI} & 1 \end{matrix}$$

where

$$x_I = x_B{*}a_{BI} + y_B{*}b_{BI} + c_{BI}$$
$$y_I = x_B{*}d_{BI} + y_B{*}e_{BI} + f_{BI}$$

Now, let us substitute the equation expressions for x_I and y_I defined by these last equations for the values x_I and y_I which were used in our earlier equations which defined x_D and y_D in terms of x_I and y_I. Doing so we obtain

$$x_D = (x_B{*}a_{BI}+y_B{*}b_{BI}+c_{BI}){*}a_{ID}+(x_B{*}d_{BI}+y_B{*}e_{BI}+f_{BI}){*}b_{ID}+c_{ID}$$
$$y_D = (x_B{*}a_{BI}+y_B{*}b_{BI}+c_{BI}){*}d_{ID}+(x_B{*}d_{BI}+y_B{*}e_{BI}+f_{BI}){*}e_{ID}+f_{ID}$$

We now collect terms, and rewrite these equations as:

$$x_D= x_B{*}(a_{BI}{*}a_{ID}+d_{BI}{*}b_{ID})+y_B{*}(b_{BI}{*}a_{ID}+e_{BI}{*}b_{ID})+(c_{BI}{*}a_{ID}+f_{BI}{*}b_{ID}+c_{ID})$$
$$y_D= x_B{*}(a_{BI}{*}d_{ID}+d_{BI}{*}e_{ID})+y_B{*}(b_{BI}{*}d_{ID}+e_{BI}{*}e_{ID})+(c_{BI}{*}d_{ID}+f_{BI}{*}e_{ID}+f_{ID})$$

We can express this last equations in matrix form as:

$$[x_D \; y_D \; 1] = [x_B \; y_B \; 1] \quad \begin{matrix} a_{BI}{*}a_{ID}+d_{BI}{*}b_{ID} & a_{BI}{*}d_{ID}+d_{BI}{*}e_{ID} & 0 \\ b_{BI}{*}a_{ID}+e_{BI}{*}b_{ID} & b_{BI}{*}d_{ID}+e_{BI}{*}e_{ID} & 0 \\ c_{BI}{*}a_{ID}+f_{BI}{*}b_{ID}+c_{ID} & c_{BI}{*}d_{ID}+f_{BI}{*}e_{ID}+f_{ID} & 1 \end{matrix}$$

From our formal notational form we have:

$$[x_D \; y_D \; 1] = [x_I \; y_I \; 1]_{ID} = [x_B \; y_B \; 1] \, \mathbf{BI \; ID}$$

Hence, we can conclude that the product **BI ID** must be defined as:

$$\mathbf{BI\;ID} = \begin{matrix} a_{BI} & d_{BI} & 0 \\ b_{BI} & e_{BI} & 0 \\ c_{BI} & f_{BI} & 1 \end{matrix} \quad \begin{matrix} a_{ID} & d_{ID} & 0 \\ b_{ID} & e_{ID} & 0 \\ c_{ID} & f_{ID} & 1 \end{matrix}$$

$$= \begin{matrix} a_{BI}*a_{ID}+d_{BI}*b_{ID} & a_{BI}*d_{ID}+d_{BI}*e_{ID} & 0 \\ b_{BI}*a_{ID}+e_{BI}*b_{ID} & b_{BI}*d_{ID}+e_{BI}*e_{ID} & 0 \\ c_{BI}*a_{ID}+f_{BI}*b_{ID}+c_{ID} & c_{BI}*d_{ID}+f_{BI}*e_{ID}+f_{ID} & 1 \end{matrix}$$

An inspection of the product matrix reveals how it is formed from its constituents. The element located in the *first row* and *first column* of the product matrix is obtained by summing the products of the elements in the *first row* of the *first matrix* by the elements in the *first column* of the *second matrix*. The element located in the *first row* and *second column* is obtained by summing the products of the elements in the *first row* of the *first matrix* by the elements in the *second column* of the *second matrix*. The general rule is that the element in the *ith row* and *jth column* of the product matrix is obtained by summing the products of the elements in the *ith row* of the *first matrix* by the elements of the *jth column* of the *second matrix*. It may be seen that, because of the presence of "0's" and "1's" in particular positions of the Interpress matrices, the generation of the product matrix involves 12 multiplications and 8 additions.

Expressed in a somewhat more formal, albeit still not rigorous manner, the rule for the multiplication of two 3 x 3 matrices proceeds as follows:

1. A 3 x 3 matrix **A** is a square array made up of nine elements. The nine elements may be viewed as consisting of three horizontal rows each containing three elements, or three vertical columns, each containing three elements. Note that the first subscript on each element is the row number, and the second subscript is the column number.

$$\mathbf{A} = \begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}$$

2. The transformation represented by two 3 x 3 matrices, **A** and **B**, applied in the order **A** first, **B** second can be represented as a third 3 x 3 matrix **C** = **B A**. If we define the three matrices as:

$$\mathbf{A} = \begin{matrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{matrix}$$

$$\mathbf{B} = \begin{matrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{matrix}$$

$$\mathbf{C} = \mathbf{B\;A} = \begin{matrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{matrix}$$

then the element c_{ij} located at the ith row and jth column of the matrix **C** is defined by the equation:

$$c_{ij} = b_{i1}a_{1j} + b_{i2}a_{2j} + b_{i3}a_{3j}$$

4. We can now observe an easy rule for the generation of c_{ij}. We take each element of the ith row of matrix **B** and multiply it by the corresponding element of the jth column of matrix A, and sum these products. The term "corresponding element" means first element multiplied by first element, second element multiplied by second element, and third element multiplied by third element. Thus, the term c_{13} is obtained by multiplying the *first* element in the first row of **B** by the first element of the *third* column of A, then multiplying the second element in the *first* row of **B** by the second element in the *third* column of A, and then multiplying the third element in the *first* row of **B** by the third element in the *third* column of A, and then summing the results of these products.

5. An example will illustrate the principle.

$$\mathbf{B} = \begin{matrix} 5 & -3 & 2 \\ -1 & 4 & 3 \\ 6 & -2 & 1 \end{matrix}$$

$$A = \begin{matrix} 4 & -1 & 5 \\ 2 & 3 & -4 \\ 1 & -4 & 2 \end{matrix}$$

First row of **B** consists of 5, -3, 2.
First column of A consists of 4, 2, 1.
c_{11} = 5x4 + -3X2 + 2x1 = 16

Second row of **B** consists of -1, 4, 3.
First column of A consists of 4, 2, 1.
c_{21} = -1x4 + 4X2 + 3x1 = 11

Third row of **B** consists of 6, -2, 1.
First column of A consists of 4, 2, 1.
c_{31} = 6x4 + -2X2 + 1x1 = 21

First row of **B** consists of 5, -3, 2.
Second column of A consists of -1, 3, -4.
c_{12} = 5x-1 + -3x3 + 2x-4 = -23

Second row of **B** consists of -1, 4, 3.
Second column of A consists of -1, 3, -4.
c_{22} = -1x-1 + 4x3 + 3x-4 = 1

Third row of **B** consists of 6, -2, 1.
Second column of A consists of -1, 3, -4.
c_{32} = 6x-1 + -2x3 + 1x-4 = -16

First row of **B** consists of 5, -3, 2.
Third column of A consists of 5, -4, 2.
c_{13} = 5x5 + -3x-4 + 2x2 = 41

Second row of **B** consists of -1, 4, 3.
Third column of A consists of 5, -4, 2.
c_{23} = -1x5 + 4x-4 + 3x2 = -15

Third row of **B** consists of 6, -2, 1.
Third column of A consists of 5, -4, 2.
c_{33} = 6x5 + -2x-4 + 1x2 = 40

Hence,

$$C = \begin{matrix} 16 & -23 & 41 \\ 11 & 1 & -15 \\ 21 & -16 & 40 \end{matrix}$$

6. If you are doing the multiplications manually, it is a simple matter to place one forefinger on the first element of the row being used in the **B** matrix, and the other forefinger on the first element of the column being used in the A matrix. Now, multiply these two values together, and move your first forefinger to the second element of the row being used in the first matrix, your second forefinger to the second element of the column being used in the second matrix, and so on, summing the products as you go.

D.4 The identity matrix and inverse matrices

There is a special matrix called the *Identity* matrix, and referred to by the letter I. The Identity matrix has the value:

$$I = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

The Identity matrix has the property that, for any matrix, A, AI = IA = A. Thus, I plays the same role in matrix multiplication that the value 1 plays in arithmetic multiplication. In conventional arithmetic, for any value a (a not equal to 0) there is another value, a^{-1}, such that a x $a^{-1} = 1$. A similar property holds for matrix operations. Under most circumstances, for any matrix A there is a matrix A^{-1} such that $AA^{-1} = I$. The matrix A^{-1} is called the *Inverse* of the matrix A.

If we have two coordinate systems represented by [x y 1] and [x' y' 1], and a matrix A such that

$$[x \;\; y \;\; 1] = [x' \;\; y' \;\; 1]\, A$$

then if we obtain A^{-1}, and multiply it on the right of both sides of the above equation we obtain

$$[x \;\; y \;\; 1]A^{-1} = [x' \;\; y' \;\; 1]\, AA^{-1} = [x' \;\; y' \;\; 1]$$

Thus, while A converts x',y' coordinates to x,y coordinates, A^{-1} converts x,y coordinates to x',y' coordinates. The matrix A^{-1} has the property that it reverses the transformation operation performed by A.

Given a general Interpress Matrix

$$A = \begin{matrix} a & d & 0 \\ b & e & 0 \\ c & f & 1 \end{matrix}$$

The matrix A^{-1} is given by

$$A^{-1} = \begin{matrix} e/D & -d/D & 0 \\ -b/D & a/D & 0 \\ (bf-ce)/D & (cd-af)/D & 1 \end{matrix}$$

where $D = ae - bd$

For example, given the matrix A

$$A = \begin{matrix} 3 & 5 & 0 \\ 1 & 2 & 0 \\ 7 & 4 & 1 \end{matrix}$$

$D = 3 \times 2 - 1 \times 5 = 1$
$bf - ce = 1 \times 4 - 7 \times 2 = -10$
$cd - af = 7 \times 5 - 3 \times 4 = 23$

Hence, A^{-1} is given by

$$A^{-1} = \begin{matrix} 2 & -5 & 0 \\ -1 & 3 & 0 \\ -10 & 23 & 1 \end{matrix}$$

Note that if $D = 0$ the quotients which include D in their denominators are undefined, and the matrix A^{-1} is undefined. As a practical matter, if the value of D is extremely small the quotients which include D in their denominators will cause arithmetic overflow in a computation process within an implementation that carries a finite number of decimal places. A matrix for which the value D is so close to zero that this condition prevails is said to be *ill-conditioned*, and does not have an inverse that can be computed within the arithmetic limitations of the processor which is attempting to carry out the inversion opertions.

D. 5 Interpress use of coordinate systems and transformations, revisited

Let us return to the application of these principles to the Interpress environment. For emphasis we repeat our earlier statement of the two critical operating principles which govern the use of transformations within Interpress 82. *The transformation parameters are stated in terms of the **old** coordinate system.* Thus, for example, if the master wishes to create a new coordinate system whose axes are rotated 30 degrees counterclockwise from those of the old coordinate system it will contain a statement which says "rotate the coordinate system +30 degrees". *The transformation matrix which is created to represent the transformation is the one which will carry coordinates expressed in the **new** coordinate system into coordinates expressed in the **old** coordinate system.* Thus, in the example stated above, the transformation will generate the matrix which will convert coordinates expressed in the rotated coordinate system into those of the unrotated coordinate system.

Transformations are created in a sequence that moves backward from the **DCS**. That is, the system starts with the **DCS**, applies the transformation that shifts to the **ICS**, then applies the transformation that shifts to the first **MCS** (designated as **BCS** in this Reader's Guide), and so on. However, all transformations are directed toward the process of producing results in the Device Coordinate System (**DCS**). That is, each transformation defines how to convert coordinates from the new coordinate system back to its predecessor coordinate system, i.e. the one that is closer to the **DCS**. Thus, for example, the transformation that shifts from the **ICS** to the **DCS** is one that will convert coordinates expressed in the **ICS** to coordinates expressed in the **DCS**. When the master then wishes to shift to the **BCS** coordinate system the transformation that is given is the one that will convert coordinates expressed in the **BCS** into coordinates expressed in the **ICS**. Thus, the statements which create the sequence of transformations move away from the DCS, first to the ICS, then to the MCS. However, the sequence of transformations which these statements create always moves towards the DCS, progressing from the BCS to the ICS, and thence from the ICS to the DCS.

One of the virtues of the matrix approach is that the transition from the DCS to the ICS, thence to the BCS, and on to any LCS can be accomplished in a series of incremental steps. Each such step involves only one simple transformation selected from a small set of fundamental transformations. The fundamental transformations with which Interpress 82 deals are the following:

1. Pure scaling in the same amount in both the x and y directions.

$$\mathbf{M} = \begin{matrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{matrix}$$

   where s is a scale factor which converts coordinates from the new coordinate system to its predecessor coordinate system.

2. Scaling by different amounts in the x and y directions.

$$\mathbf{M} = \begin{matrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{matrix}$$

   where s_x and s_y are scale factors which convert coordinates from the new coordinate system to its predecessor coordinate system.

3. Pure rotation.

$$\mathbf{M} = \begin{matrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{matrix}$$

   where *a* is positive for the case where it requires a counterclockwise rotation of the *predecessor* coordinate axes to obtain the new coordinate system. Note that the definition of the angle a is given in terms of the sign of the rotation that would be required to bring the predecessor axes to the new coordinate system, but the transformation given is the one that converts coordinates in the *new* coordinate system to its predecessor coordinate system.

4. Mirror Image about x-axis.

$$\mathbf{M} = \begin{matrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

5. Mirror Image about y-axis.

$$\mathbf{M} = \begin{matrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

6. Pure translation.

$$\mathbf{M} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{matrix}$$

   where T_x and T_y are the translation components, expressed in the predecessor coordinate system, which would be required to bring the origin of the *predecessor* axes to the origin of the new coordinate system, but the transformation given is the one that converts coordinates in the *new* coordinate system to its predecessor coordinate system.

(The following warning comment is for the mathematically knowledgeable reader who has a strong prior background in transformations and matrix operations. Note that each of the above transformations describes its components in terms of the *predecessor* coordinate system, but the transformation which is generated is the transformation of coordinates from the *new* coordinate system *to* the predecessor coordinate system. In mathematical terms then, the transformations stated above are the *inverses* of the transformations of conventional mathematics. Thus, for example, the definition of a rotation is in terms of the angle through which the predecessor coordinate system must be rotated in order to bring it into alignment with the new coordinate system. The transformation which is generated by the definition of the rotation operation is one which will convert coordinates in the new coordinate system into the predecessor coordinate system.)

D.6 The concatenation principle in Interpress 82

The introduction of a new coordinate system in Interpress is accomplished by providing the transformation which will transform coordinates expressed in this new coordinate system into the coordinates expressed in its predecessor coordinate system. Two important properties must be observed if this is to be accomplished correctly. These are:

1. The creator of the new coordinate system thinks in terms of the predecessor coordinate system in describing the parameters of the required transformation. The transformation generated as a result of that description is one which will carry the coordinates of the new coordinate system into those of the predecessor coordinate system.
2. The matrix representing the predecessor transformation must be left-multiplied by the generated transformation. In Interpress 82 this process is referred to as concatenation, and *always* has the meaning of left concatenation.

A simple example will illustrate the principle. Assume that the printer has a resolution of 300 black-or-white pixels per inch, and that the printer coordinate system uses 1/300th inch as its unit. The Interpress Coordinate System uses meters as its unit. The first step in getting from the Device Coordinate System to the Interpress Coordinate System is to effect a scaling transformation which will place the system into the Interpress Coordinate System's units. One meter is equal to 39.37 inches, and 39.37 inches is equal to 39.37 x 300 1/300th inches. Therefore one meter is equal to 11811 1/300ths inches. If we wish to shift the coordinate system from the Device Coordinate System units to the Interpress Coordinate System units we must express the transformation which will convert meters to 1/300ths inches. In Interpress operators this is stated as 11811 SCALE CONCATT. The operation 11811 SCALE produces the matrix

$$\mathbf{M} = \begin{matrix} 11811 & 0 & 0 \\ 0 & 11811 & 0 \\ 0 & 0 & 1 \end{matrix}$$

This matrix is then concatenated (left multiplied) with the current transformation. Note that the transformation expresses the conversion of units in the Interpress Coordinate System into units in the Device Coordinate System.

In Interpress 82 there are two concatenation operators, named CONCAT and CONCATT, respectively. CONCAT requires that two transformations (i.e. transformation matrices) be on the top of the stack at the time it is invoked. If we denote them by **M** and **N**, with **M** having been pushed on the stack first, **N** second, then the operator CONCAT forms the transformation defined by the matrix product **MN**. In execution CONCAT pops the stack to obtain **N**, pops the stack again to obtain **M**, performs the matrix multiplication **MN** (in that order), and places the product matrix back on the stack. CONCATT requires that one transformation be on the top of the stack at the time it is invoked. If we denote that transformation by **M**, and the current transformation (held in the Imager State) by **T**, then the operator CONCATT forms the transformation defined by the matrix product **MT**, and establishes that product as the value of the current transformation in the Imager State. In execution CONCATT obtains the matrix **T** representing the current transformation from the Imager State, pops the stack to obtain **M**, performs the matrix multiplication **MT** (in that order), and establishes the product matrix as the current value of **T** in the Imager State.

D.7 Illustrative example

The following paragraphs will illustrate the principles presented above in terms of a typical Interpress 82 example. For purposes of this example we make the following assumptions (See Figure D.1):

1. The printer utilizes a right-handed coordinate system with origin in the lower left hand corner of a portrait page, y-axis parallel to the long edge of the paper, x-axis parallel to the short edge of the paper. The printer has a resolution of 1/300 inches and uses 1/300 inches as its units of measurement.
2. The Interpress Coordinate System is as defined in the Interpress 82 Specification, namely, a right-handed coordinate system with origin in the lower left hand corner of a portrait page, y-axis parallel to the long edge of the paper, x-axis parallel to the short edge of the paper. The units of the Interpress 82 Coordinate System are meters.
3. The master's Base Coordinate System is a right-handed coordinate system with origin in the lower left hand corner of a landscape page, y-axis parallel to the short edge of the paper, x-axis parallel to the long edge of the paper. The master coordinate system uses inches as its unit of measurement.

Let us now follow the matrix generation process at the printer. It is important to keep in mind the fact that the goal of the transformation process is to bring coordinates expressed in any coordinate system into coordinates expressed in the Device Coordinate System. It is also important to keep in mind that the creator of the master has no knowledge whatsoever of the Device Coordinate System. He must express everything in terms of the Interpress Coordinate System. The printer itself will take care of the final transformation from the Interpress Coordinate System to the Device Coordinate System. The creator of the master must also keep in mind the Interpress view of transformations. Thus, if it requires a number of matrix multiplications to convert from his Master Coordinate System to the Interpress Coordinate System he must carry them out by a series of operations which work *backwards* from the Interpress Coordinate System out to his Master Coordinate System. Each step in this process *backs out* from the Interpress Coordinate System, but describes how to get from the coordinate system thus achieved *back to* its predecessor. If we follow the sequence of operations this process will become clear. This sequence is illustrated in Figure D.1 which should be referred to as the discussion progresses.

The current transformation, T, is initialized to the Identity matrix, I, at the beginning of the document. Thus, we have, initially:

$$T = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

The first thing the printer will do will be to generate the matrix which will convert Interpress Coordinates into Device Coordinates. Recall that it does so by concatenating the transformation **ID** (Interpress to Device) with the identity matrix to form a new value for the Current Transformation, T, i.e. it generates T = **ID** * **I** = **ID** by an internally generated **ID** CONCATT sequence. (The creator of the master is totally unaware of this operation, and is indifferent to it, because he only has the responsibility to create the transformation which will take his initial Master Coordinate System into the Interpress Coordinate System. In fact, this operation will generally be different in each printer because of differences in implementation and resolution.)

The printer is going to back out from its Device Coordinate System to the Interpress Coodinate System, hence it is required to create the transformation which will carry the Interpress Coordinate System back to the Device Coordinate System. Since, in our example, the Device Coordinate System and the Interpress Coordinate System are identical except for the units which each uses this transformation will merely be the scale change previously described.

Thus, the transformation **ID** (Interpress to Device) must be the transformation generated by:

$$11811 \text{ SCALE} = \begin{matrix} 11811 & 0 & 0 \\ 0 & 11811 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Hence the operation at the printer becomes 11811 SCALE CONCATT, which results in the operation:

$$T = \begin{matrix} 11811 & 0 & 0 \\ 0 & 11811 & 0 \\ 0 & 0 & 1 \end{matrix} \qquad \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

and produces the matrix:

$$T = \begin{matrix} 11811 & 0 & 0 \\ 0 & 11811 & 0 \\ 0 & 0 & 1 \end{matrix}$$

We must now generate the series of transformations which takes us from the Interpress Coordinate System back out to the Master Coordinate System. We do so by means of three successive incremental transformations. First we create an intermediate coordinate system with the same origin and orientation as the Interpress Coordinate System, but having the units of the Master Coordinate System. We are going to back the Interpress Coordinate System out to this first intermediate coordinate system, but in doing so we create the transformation which will return coordinates from this first intermediate coordinate system to those of the Interpress Coordinate System. Again, this transformation is merely a scale change.

Now, 1 inch = .0254 meters.

Hence, the intermediate coordinate system coordinates of a point would have to be multiplied by .0254 to obtain the Interpress coordinates of the same point.

Thus, the desired transformation is

$$.0254 \text{ SCALE} = \begin{matrix} 0.0254 & 0 & 0 \\ 0 & 0.0254 & 0 \\ 0 & 0 & 1 \end{matrix}$$

When CONCATT is applied we obtain:

$$T = \begin{matrix} .0254 & 0 & 0 \\ 0 & .0254 & 0 \\ 0 & 0 & 1 \end{matrix} \quad \begin{matrix} 11811 & 0 & 0 \\ 0 & 11811 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Hence,

$$T = \begin{matrix} 300 & 0 & 0 \\ 0 & 300 & 0 \\ 0 & 0 & 1 \end{matrix}$$

This last transformation makes sense because it concatenates the scaling from inches to meters with the scaling from meters to 1/300ths inches. The result clearly must be the conversion from inches to 1/300ths inches, and there are clearly 300 1/300ths of an inch per inch.

Now we create a second intermediate coordinate system whose origin is translated +8.5 inches along the x-axis from the origin of the first intermediate coordinate system. Note that in order to convert the coordinates of a point in the second intermediate coordinate system into the coordinates of a point in the first intermediate coordinate system it is necessary to add 8.5 to the x-coordinate (e.g. the point (0,0) in the second intermediate coordinate system is the point (8.5, 0) in the first intermediate coordinate system. Therefore, the Interpress expression for the transformation which will carry coordinates expressed in this second intermediate coordinate system back to the first intermediate coordinate system is +8.5 0 TRANSLATE, and the desired transformation is:

$$+8.5 \ 0 \text{ TRANSLATE} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ +8.5 & 0 & 1 \end{matrix}$$

We now perform a CONCATT operation to produce:

$$T = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ +8.5 & 0 & 1 \end{matrix} \quad \begin{matrix} 300 & 0 & 0 \\ 0 & 300 & 0 \\ 0 & 0 & 1 \end{matrix}$$

Hence,

$$T = \begin{matrix} 300 & 0 & 0 \\ 0 & 300 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

Finally, we create the Base Coordinate System by rotating the second intermediate coordinate system. Note that it requires a 90 degree counterclockwise rotation (+90 degrees) of the second intermediate coordinate system to arrive at the Base Coordinate System. Therefore the Interpress expression for the transformation that will carry coordinates from the Base Coordinate System to the second intermediate coordinate system is 90 ROTATE, and the desired transformation is:

$$90\ \text{ROTATE} = \begin{matrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}$$

We now perform a CONCATT operation to produce:

$$T = \begin{matrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix} \quad \begin{matrix} 300 & 0 & 0 \\ 0 & 300 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

Hence,

$$T = \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

This transformation is one that will carry a set of coordinates expressed in units of inches in the Base Coordinate System to the coordinates of the same point expressed in 1/300ths inches in the Device Coordinate System. For example, consider the point 5,2 in the Base Coordinate System, and apply the Current Transformation, T, to it thus:

$$\begin{matrix} 5 & 2 & 1 \end{matrix} \quad \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

The product of the one-row matrix by the transformation T is:

$$\begin{matrix} 1950 & 1500 & 1 \end{matrix}$$

This corresponds to a point which is located 1950 1/300ths inches from the lower left hand corner of the paper along the short edge, and 1500 1/300ths inches up from the bottom of the page on a line parallel to the long edge of the paper. Converting to inches produces the point 6.5 inches, 5.0 inches. Thus, it corresponds to a point which is 6.5 inches from the lower left hand corner of the paper along the short edge, and 5.0 inches up from the bottom of the page on a line parallel to the long edge of the paper, and that is precisely correct.

In summary, then, the sequence of Interpress operations which would appear in the master to create the Base Coordinate System is:

```
.0254 SCALE
CONCATT
8.5 0 TRANSLATE
CONCATT
90 ROTATE
CONCATT
```

Now, let us assume that the creator of the master wishes to establish a local coordinate system whose origin is at the point 3,4 in the Base Coordinate System, and whose axes are rotated 30 degrees counter-clockwise from those of the Base Coordinate System. **(Note:** A 30 degree rotation is not permissible in Interpress 82. This example is included to demonstrate the full power of the transformation operations. It would be rejected by an Interpress 82 printer as an inadmissible operation.) The transformation which will transform coordinates expressed in this local coordinate system to the Device Coordinate system is established by concatenating two additional transformations to the transformation **T** which currently carries the initial Master Coordinates into the Device Coordinates. The Interpress expression for the transformation which will carry coordinates expressed in a coordinate system whose axes are parallel to the Base Coordinate System, but whose origin is at the point 3,4 in the Base Coordinate system back to the Base Coordinate System is 3 4 TRANSLATE, represented by:

$$
3\ 4\ \text{TRANSLATE} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 4 & 1 \end{matrix}
$$

We now perform a CONCATT operation to produce:

$$
\text{T} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 4 & 1 \end{matrix} \quad \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 2550 & 0 & 1 \end{matrix}
$$

which multiplies out to produce:

$$
\text{T} = \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 1350 & 900 & 1 \end{matrix}
$$

We now generate a second local coordinate system whose origin is coincident with that of this first local coordinate system, but whose axes are rotated 30 degrees counter-clockwise from it. The Interpress expression that will provide the transformation that will carry coordinates expressed in this second local coordinate system back to the first local coordinate system is given by 30 ROTATE, and is represented by:

$$
30\ \text{ROTATE} = \begin{matrix} \cos 30 & \sin 30 & 0 \\ -\sin 30 & \cos 30 & 0 \\ 0 & 0 & 1 \end{matrix}
$$

which is expressed numerically as:

$$\text{30 ROTATE} = \begin{matrix} .8660254 & .5 & 0 \\ -.5 & .8660254 & 0 \\ 0 & 0 & 1 \end{matrix}$$

We now perform a CONCATT operation to produce:

$$T = \begin{matrix} .8660254 & .5 & 0 \\ -.5 & .8660254 & 0 \\ 0 & 0 & 1 \end{matrix} \quad \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 1350 & 900 & 1 \end{matrix}$$

which multiplies out to produce:

$$T = \begin{matrix} -150 & 259.808 & 0 \\ -259.808 & -150 & 0 \\ 1350 & 900 & 1 \end{matrix}$$

Finally, consider the point 2,1 expressed in this last local Master Coordinate System, and apply the current transformation, T, to it thus:

$$\begin{matrix} 2 & 1 & 1 \end{matrix} \quad \begin{matrix} -150 & 259.808 & 0 \\ -259.808 & -150 & 0 \\ 1350 & 900 & 1 \end{matrix}$$

The product of the one-row matrix by the transformation T is:

$$\begin{matrix} 790.192 & 1269.616 & 1 \end{matrix}$$

This corresponds to a point which is located 790.192 1/300ths inches from the lower left hand corner of the paper along the short edge, and 1269.616 1/300ths inches up from the bottom of the page on a line parallel to the long edge of the paper. Converting to inches produces the point 2.634 inches, 4.232 inches. Thus, it corresponds to a point which is 2.634 inches from the lower left hand corner of the paper along the short edge, and 4.232 inches up from the bottom of the page on a line parallel to the long edge of the paper, and that is precisely correct. The exact expressions for the location of the referenced point, expressed in 1/300ths inches, is:

x = 300*(3 + sqrt(5)*(cos($\tan^{-1}$0.5 + 30))) = 1269.615 1/300ths inches

y = 300*(8.5 - (4 + sqrt(5)*(sin($\tan^{-1}$0.5 + 30)))) = 790.192 1/300ths inches.

In summary, then, the sequence of Interpress operations which would appear in the master to create the final local Master Coordinate System is:

.0254 SCALE
CONCATT
8.5 0 TRANSLATE
CONCATT
90 ROTATE
CONCATT
3 4 TRANSLATE
CONCATT
30 ROTATE
CONCATT

D.8 Normalizing pixel arrays

A scanned image of an area xPixels by yPixels is defined in a Standard Coordinate System using the following conventions.

1. The first pixel on the first scan line is located at the origin.
2. The subsequent pixels on the first scan line proceed up the y-axis, with the last pixel on the first scan line being located at 0, yPixels-1.
3. The first pixel of the second scan line is located at (1,0), and subsequent pixels on the second scan line proceed up the line x = 1, with the last pixel on the second scan line being located at (1, yPixels-1).
4. The first pixel on the last scan line is located at (xPixels-1,0), and subsequent pixels on the last scan line proceed up the line x = xPixels-1, with the last pixel on the last scan line being located at (xPixels-1, yPixels-1).

Thus, the orientation of an image as it appears in the Standard Coordinate System is a function of the starting point of the scan, the direction of the "fast" scan, (i.e. which direction each scan line goes), and the direction of the "slow" scan, (i.e. the direction in which the sequential scan lines appear). Figure D.2 shows the orientations of a large letter "J" as it would appear in the SCS for the eight possible scanning sequences.

The definition of a *PixelArray* includes a transformation, *m*, which "normalizes" the scanned image. A scanned image is said to be normalized when it has the following characteristics:

1. The image appears in its "desired" viewing position (i.e. the position in which the image user desires it to be viewed. If I wish you to view an upside down "J" then upside down is its desired viewing position.)
2. The origin is in the lower left hand corner of the desired viewing position.
3. The upper left hand corner of the image in the desired viewing position is at the point (0,1), i.e. the image is scaled so that it occupies a rectangle whose height is one unit.

Figure D.2 exhibits the Interpress operator sequence together with the resultant transformation matrix for normalizing each of the eight possible scanning sequences under the assumption that the desired viewing position is the conventional upright position.

An example will illustrate how each of these normalizing transformations is generated. Consider the case shown in the lower left hand of Figure D.2. The scanning sequence is assumed to start in the upper left hand corner of the rectangle bounding the "J". The fast scan is from left to right, the slow scan from the top to bottom. The figure which results in the SCS is a counterclockwise rotated "J" with its smaller leg resting on the x-axis, and the top of its longer leg lying on the y-axis. We use the same technique we have employed in the past to "back" the MCS coordinate system into this configuration. Consider that we have the "J" upright in the MCS with origin at its lower left hand corner, and the upper left hand corner of its bounding rectangle located at (0,1). Now, perform the operation 0 1 TRANSLATE. This creates a coordinate system with origin at (0,1), and provides the transformation that will convert coordinates from that system to the MCS from which we started. Next, perform the operations -90 ROTATE, and left concatenate it with its predecessor transformation. This creates a coordinate system that is properly rotated with respect to its predecessor, and concatenates this transformation with its predecessor. The resultant transformation is one that will carry coordinates from the translated/rotated coordinate system back to the **MCS**. Finally perform the operations 1/xPixels SCALE, and left concatenate it with its predecessor transformation. This creates a coordinate system whose units are 1/xPixels times those of the **MCS**, and provides the final transformation that will carry coordinates from the translated/rotated/scaled coordinate system back to the **MCS**. The Interpress 82 statement of this sequence of operations is:

```
1/xPixels SCALE
-90 ROTATE
CONCAT
0 1 TRANSLATE
CONCAT
```

In matrix terms the matrix algebra proceeds as follows:

$$\text{1/xPixels SCALE} = \begin{matrix} 1/\text{xPixels} & 0 & 0 \\ 0 & 1/\text{xPixels} & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\text{-90 ROTATE} = \begin{matrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\text{CONCAT} = \begin{matrix} 0 & -1/\text{xPixels} & 0 \\ 1/\text{xPixels} & 0 & 0 \\ 0 & 0 & 1 \end{matrix}$$

$$\text{0 1 TRANSLATE} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{matrix}$$

$$\text{CONCAT} = \begin{matrix} 0 & -1/\text{xPixels} & 0 \\ 1/\text{xPixels} & 0 & 0 \\ 0 & 1 & 1 \end{matrix}$$

The reader can confirm that the resultant transformation performs the desired normalization.

D.9 Pixel array example

Consider the case where we would wish to print the enlarged image of the capital letter "J" shown in Figure D.2 on a page described within the coordinate frameworks of our previous example. Although the **MCS** has its origin at the lower left hand corner of a landscape page, the **DCS** has its origin at the upper left hand corner of a landscape page. In most printer implementations the image scan sequence must match the printer scan sequence. If we assume this to be the case in this instance then the image must be scanned beginning in the upper left hand corner of its enclosing rectangle, fast scan left-to-right, slow scan top-to-bottom. The orientation of such a scanned image in the **SCS** is shown in the lower left hand corner of Figure D.2.

Let us assume that the image of the "J" is 3 inches wide by 4 inches high, and that it is scanned at a resolution of 150 black-white pixels per inch in both the fast- and slow-scan directions. Let us further assume that we wish to create an image that is also 3 inches wide by 4 inches high, with its lower left hand corner located at the point (1,2) in the **BCS**.

As Figure D.2 shows, the transformation matrix representing the normalizing transformation, ***m***, for this scanned image is given by:

$$\boldsymbol{m} = \begin{matrix} 0 & -1/\text{xPixels} & 0 \\ 1/\text{xPixels} & 0 & 0 \\ 0 & 1 & 1 \end{matrix}$$

Because the original image was scanned at 150 bits per inch, its 3 inch by 4 inch size generates a scanned array of 450 pixels by 600 pixels. The 600 pixel dimension lies along the x-axis in the SCS, hence xPixels = 600, and the matrix ***m*** for this specific instance becomes:

$$\boldsymbol{m} = \begin{matrix} 0 & -1/600 & 0 \\ 1/600 & 0 & 0 \\ 0 & 1 & 1 \end{matrix}$$

Now, from our previous development we know that the value of the current transformation, **T**, that will carry the **BCS** to the **DCS** is given by:

$$\mathrm{T} = \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

In order to image our character at the point (1,2) we must perform the operation 1 2 TRANSLATE CONCATT, which results in:

$$\mathrm{T} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 2 & 1 \end{matrix} \quad \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

$$\mathrm{T} = \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 1950 & 300 & 1 \end{matrix}$$

Since our normalized character is one unit high, and, in this case, our unit is an inch, and we wish it to be 4 inches high, we must now perform the operation 4 SCALE CONCATT, which results in:

$$\mathrm{T} = \begin{matrix} 4 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 1 \end{matrix} \quad \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 1950 & 300 & 1 \end{matrix}$$

$$\mathrm{T} = \begin{matrix} 0 & 1200 & 0 \\ -1200 & 0 & 0 \\ 1950 & 300 & 1 \end{matrix}$$

Finally, we concatenate ***m*** with T to obtain the transformation which applies when we image our Pixel Array, which results in:

$$\boldsymbol{m}\mathrm{T} = \begin{matrix} 0 & -1/600 & 0 \\ 1/600 & 1 & 0 \\ 0 & 1 & 1 \end{matrix} \quad \begin{matrix} 0 & 1200 & 0 \\ -1200 & 0 & 0 \\ 1950 & 300 & 1 \end{matrix}$$

$$\boldsymbol{m}\mathrm{T} = \begin{matrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 750 & 300 & 1 \end{matrix}$$

Let us interpret this matrix, and demonstrate that it is correct. Consider the general coordinates (X_s,Y_s) of a pixel in the **SCS**. To obtain the device coordinates of this point we transform its vector representation by the matrix ***m*T**. Thus:

$$[X_D \ \ Y_D \ \ 1] = [X_S \ \ Y_S \ \ 1] \ \boldsymbol{m}\mathrm{T}$$

$$[X_D \ \ Y_D \ \ 1] = [X_S \ \ Y_S \ \ 1] \quad \begin{matrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 750 & 300 & 1 \end{matrix}$$

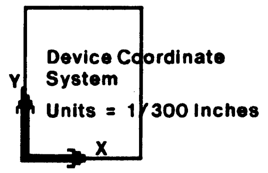
If we expand this latter form into its equation form we obtain:

$$X_D = 2X_S + 750$$

$$Y_D = 2Y_S + 300$$

From this form it is clear that the origin of the pixel array in the **SCS** goes to the point (750,300) in the **DCS**. If these coordinates were scaled to inches they would be (2.5,1) since there are 300 **DCS** units per inch. From Figure D.1 it should be clear that the x-coordinate of **DCS** (stated in inches) is 8.5 minus the y-coordinate of **BCS**, and the y-coordinate of **DCS** is equal to the x-coordinate of **BCS**. Now we wanted a 4 inch high image with lower left hand corner located at the point (1,2) in the landscape page viewpoint of **BCS**. If we add the 4 inch height of the image to the 2 inch **BCS** y-coordinate we obtain the **BCS** y-coordinate value of 6 for the upper left hand corner of the image. Such a value is equivalent to a DCS x-coordinate value of 8.5 - 6 = 2.5 inches. Similarly the 1 inch x-coordinate in the **BCS** corresponds to the 1 inch y-coordinate in the **DCS**. Hence the coordinates of the upper left hand corner of the scanned image (its origin in this scanning process) is properly located.

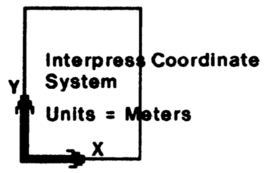
We can also see that each pixel of the first scan line will be deposited on every other pixel along the corresponding scan line in the output image. That is, the pixels of the first scan line will be deposited at locations (750,300), (750,302), (750,304), and so on. Pixels on the second scan line will be deposited at locations (752,300), 752,302), (752,304), and so on. Note that the printer is assumed to print at 300 bits per inch while the image was scanned at only 150 bits per inch. If the factor of 2 were not present the 600 pixels along the height of the image would be printed on 600 consecutive printer pixels which would only occupy 2 inches on the printed page. In the absence of any special hardware/firmware/software combination in the printer the factor of 2 would cause the 600 pixels along a scan line of the image to be deposited on every other pixel as shown above. However, most printer implementations will fill in the missing pixels by means of some interpolation process. In this case a simple bit replication along the scan line, and a scan line replication of each scan line provides an example of a possible interpolation process.



Interpress to Device Transformation

11811 SCALE — CONCATT

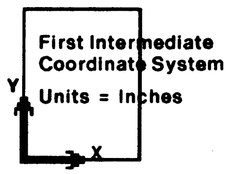
$$\begin{matrix} 11811 & 0 & 0 \\ 0 & 11811 & 0 \\ 0 & 0 & 1 \end{matrix} \qquad \begin{matrix} 11811 & 0 & 0 \\ 0 & 11811 & 0 \\ 0 & 0 & 1 \end{matrix}$$



First Intermediate to Interpress Transformation

254/10000 SCALE — CONCATT

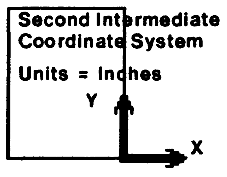
$$\begin{matrix} .0254 & 0 & 0 \\ 0 & .0254 & 0 \\ 0 & 0 & 1 \end{matrix} \qquad \begin{matrix} 300 & 0 & 0 \\ 0 & 300 & 0 \\ 0 & 0 & 1 \end{matrix}$$



Second Intermediate to First Intermediate Transformation

85/10 0 TRANSLATE — CONCATT

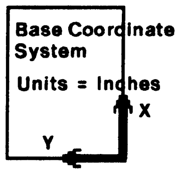
$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 8.5 & 0 & 1 \end{matrix} \qquad \begin{matrix} 300 & 0 & 0 \\ 0 & 300 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

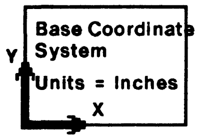


Base to Second Intermediate Transformation

90 ROTATE — CONCATT

$$\begin{matrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix} \qquad \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 2550 & 0 & 1 \end{matrix}$$

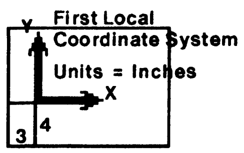


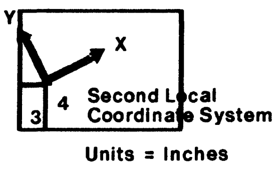


First Local to Base Transformation

3 4 TRANSLATE — CONCATT

$$\begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 4 & 1 \end{matrix} \qquad \begin{matrix} 0 & 300 & 0 \\ -300 & 0 & 0 \\ 1350 & 900 & 1 \end{matrix}$$





Second Local to First Local Transformation

30 ROTATE — CONCATT

$$\begin{matrix} 0.8660254 & 0.5 & 0 \\ -0.5 & 0.8660254 & 0 \\ 0 & 0 & 1 \end{matrix} \qquad \begin{matrix} -150 & 259.808 & 0 \\ -259.808 & -150 & 0 \\ 1350 & 900 & 1 \end{matrix}$$

Figure D.1 Coordinate Transformation Sequence

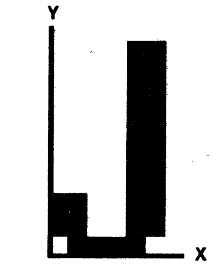
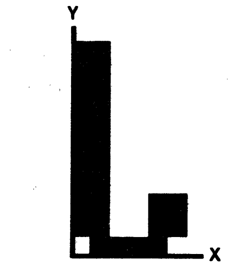
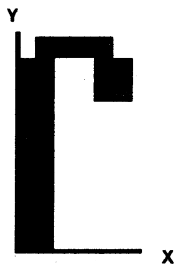
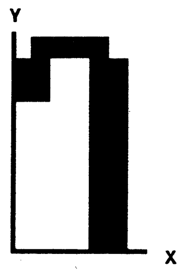
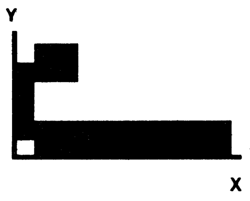
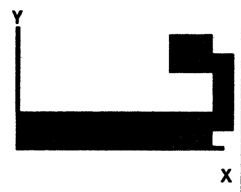
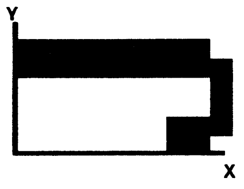
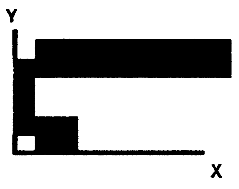
1/yPixels SCALE $\begin{matrix} \frac{1}{yPixels} & 0 & 0 \\ 0 & \frac{1}{yPixels} & 0 \\ 0 & 0 & 1 \end{matrix}$	 Fast -- Bottom to Top Slow -- Left to Right	 Fast -- Bottom to Top Slow -- Right to Left	1/yPixels SCALE -1 1 SCALE2 CONCAT xPixels/yPixels 0 TRANSLATE CONCAT $\begin{matrix} \frac{-1}{yPixels} & 0 & 0 \\ 0 & \frac{1}{yPixels} & 0 \\ \frac{xPixels}{yPixels} & 0 & 1 \end{matrix}$
1/yPixels SCALE 1 -1 SCALE2 CONCAT -1 1 SCALE2 CONCAT xPixels/yPixels 1 TRANSLATE CONCAT $\begin{matrix} \frac{-1}{yPixels} & 0 & 0 \\ 0 & \frac{-1}{yPixels} & 0 \\ \frac{xPixels}{yPixels} & 1 & 1 \end{matrix}$	 Fast -- Top to Bottom Slow -- Right to Left	 Fast -- Top to Bottom Slow -- Left to Right	1/yPixels SCALE 1 -1 SCALE2 CONCAT 0 1 TRANSLATE CONCAT $\begin{matrix} \frac{1}{yPixels} & 0 & 0 \\ 0 & \frac{-1}{yPixels} & 0 \\ 0 & 1 & 1 \end{matrix}$
1/xPixels SCALE xPixels/yPixels 0 TRANSLATE CONCAT 90 ROTATE CONCAT $\begin{matrix} 0 & \frac{1}{xPixels} & 0 \\ \frac{-1}{xPixels} & 0 & 0 \\ \frac{xPixels}{yPixels} & 0 & 1 \end{matrix}$	 Fast -- Right to Left Slow -- Bottom to Top	 Fast -- Right to Left Slow -- Top to Bottom	1/xPixels SCALE 1 -1 SCALE2 CONCAT -90 ROTATE CONCAT xPixels/yPixels 1 TRANSLATE CONCAT $\begin{matrix} 0 & \frac{-1}{xPixels} & 0 \\ \frac{-1}{xPixels} & 0 & 0 \\ \frac{xPixels}{yPixels} & 1 & 1 \end{matrix}$
1/xPixels SCALE -90 ROTATE CONCAT 0 1 TRANSLATE CONCAT $\begin{matrix} 0 & \frac{-1}{xPixels} & 0 \\ \frac{1}{xPixels} & 0 & 0 \\ 0 & 1 & 1 \end{matrix}$	 Fast -- Left to Right Slow -- Top to Bottom	 Fast -- Left to Right Slow -- Bottom to Top	1/xPixels SCALE -1 1 SCALE2 CONCAT -90 ROTATE CONCAT $\begin{matrix} 0 & \frac{1}{xPixels} & 0 \\ \frac{1}{xPixels} & 0 & 0 \\ 0 & 0 & 1 \end{matrix}$

Figure D.2 Pixel Arrays in the Standard Coordinate System for all Possible Scanning Sequences

Normalizing Operations and Resultant Matrices

Index

XEROX

Xerox Corporation
El Segundo, California 90245

Printed in U.S.A.

610P72580