XEROX

Interpress Electronic Printing Standard

Interpress Electronic Printing Standard

Xerox System Integration Standard

XNSS 048601

Xerox
System Integration
Standard

INTERPRESS
ELECTRONIC PRINTING STANDARD

XEROX

XNSS 048601
January 1986

Notice

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

This document is being provided for informational purposes only. Xerox makes no warranties or representations of any kind relative to this document or its use, including the implied warranties of merchantability and fitness for a particular purpose. Xerox does not assume any responsibility or liability for any errors or inaccuracies that may be contained in the document, or in connection with the use of this document in any way.

The information contained herein is subject to change without any obligation of notice on the part of Xerox.

This document was produced using the Interpress 3.0 standard. Text and graphics were created on the Xerox 8010 and 6085 Professional Computer Systems using the ViewPoint software. The figures in Chapter 4, Imaging Operators, were created using Interpress 3.0 instructions directly. The camera-ready copy was produced on a very high-resolution Interpress laser printer at Xerox.

The document is available in Interpress, using the Commercial Set, for printing on any of the Xerox Interpress network laser printers including the Xerox NS 8000 LaserCP, 8044, 4050, 8700, and 9700.

Printed in U.S.A.

PREFACE

This publication is one of a family of publications that collectively describe the standards underlying Xerox Printing Systems.

The Interpress Electronic Printing 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.

This publication defines and explains the Interpress standard, gives examples of its use, explains how to create documents in Interpress, and explains how a raster printer goes about printing documents that are encoded in the standard. The primary purpose of this publication is to provide an accurate specification of the Interpress standard.

This publication supersedes the Interpress 82 Electronic Printing Standard (XSIS 048201), the Interpress Electronic Printing Standard, Version 2.0 (XSIS 048306), and the Interpress Electronic Printing Standard, Version 2.1 (XSIS 048404). Significant differences between these publications and the current standard are summarized in Appendix D.

This publication (XNSS 048601) differs from the previous publication (XNSS 048512) only in minor details of formatting and editorial improvements.

Comments and suggestions on this publication 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

LIST OF FIGURES AND TABLES

Figures:

Tables:

1. INTRODUCTION

This specification is a rather formal description of the Interpress™ electronic printing standard. Interpress defines a digital representation for documents which can be printed on a variety of electronically controlled printers, most notably on raster printers. A document in Interpress form is called an *Interpress master*. Like an offset or mimeograph master, an Interpress digital master can be used to produce any number of copies of the document it represents. An Interpress master is made by a program called the *creator*. The master can then be stored in a file for later demand printing, transmitted to other sites as a means of communicating complete documents, or printed on different printers at the same site when there are needs for varying quality and speed.

An Interpress master precisely describes the desired or ideal appearance of *a document that has been completely composed by some other process.* All line ending, hyphenation, and line justification decisions, and in fact all decisions about the shapes and positions of the images, are made before creating the master. Since Interpress describes a document in a device-independent manner, a master can be printed on a variety of devices, each of which renders its best approximation to the ideal represented by the master.

The purpose of this specification is to describe precisely, clearly, and concisely the form and meaning of an Interpress master. Precision demands a rather formal style of description, which can be difficult to grasp on first reading. Companion reports contain commentary, tutorial, and explanatory material intended to help readers in understanding Interpress (*Introduction to Interpress and Interpress Reader's Guide*). If you are learning about Interpress for the first time, start with the commentary. This specification, however, remains the final authority on the definition of Interpress.

Here and there throughout this specification there are paragraphs of fine print, like this. Material that is not needed to specify the standard formally is in fine print. This material may be examples, hints on how to use features, redundant explanations, or any other information which is auxiliary to the standard itself.

Masters that specify relatively simple images (such as the pages of this document) need only some of the facilities of Interpress. Sections of the specification flagged with a dagger character (†) describe facilities which are not needed for such masters.

The specification contains a number of programs which define certain aspects of Interpress or provide examples of its use. In these programs, comments are enclosed in "--" brackets:

-- *This is a comment.* --

2. THE BASE LANGUAGE

This chapter defines the *base language* in which Interpress masters are expressed. The base language contains no facilities for output. Instead, it provides a framework within which additional primitive operators can be invoked; using these output primitives and the facilities of the base language, an Interpress master can specify images, character sets, or other things. The structure of an Interpress master and the possible interactions between a master and the external world are described in Chapter 3. Operators and types for image output are described in Chapter 4.

2.1 Introduction

Interpress can be used to specify a very wide variety of images with a high degree of device independence. To provide this power without too much special-purpose mechanism requires a programming language. To make this language both concise and adaptable, there are general ways to:

- structure data (vectors),
- define procedures (composed operators) with local variables (frames),
- limit the effects of calling a procedure (stack marks, DOSAVE, and DOSAVEALL).

Masters which specify simple images do not need these facilities. They use only the parts of the base language described in the following sections:

§2.2.1-2	Numbers and identifiers
§2.3.1	The stack
§2.4	Operator notation, summary of shorthands
§2.4.1	Errors

The reader may wish to skip the other sections (marked with a †) on first reading.

2.2 Types and literals

The Interpress base language manipulates *values*. With two exceptions (frame elements and imager variables), these values are *constant* and cannot be changed once they are constructed. Except for the imager variables, there is no *sharing* of data in Interpress; values are always transmitted by copying.

Of course the implementation need not actually do copying. The elements of a vector, for example, have fixed values which are determined when the vector is constructed; an Interpress vector is like a Pascal **const** array in this respect, and unlike a Pascal **var** array or a Fortran or Basic array. It differs from an array in most programming languages in that the elements need not all have the same type.

Every value has a *type*. There are six types in the base language: Number, Identifier, Mark, Vector, Body, and Operator. In addition, there are two other types which describe the values required by certain operators:

- *Any* is a type which accepts a value of any type except Body or Mark.
- *Cardinal* values are a subset of Number values.

There are Number, Identifier, Body, and Operator *literals* which denote certain values of these types. The set of literals in the language is defined by giving a specific syntax for each kind of literal, together with a mapping from this syntax into values of the type. The actual representation of literals in an Interpress master, however, is defined by the encoding specified in §2.5.

Other types are defined for the imaging operators (Chapter 4); there are no literals of these types.

The printer may place *limits* on the sizes of various values. The minimum values of these limits are defined in §5.6.1.

2.2.1 Numbers and Cardinals

A *Number* is an element of a certain subset of the rationals. This subset must contain every integer in the range of $-(2^{24}-1)...(2^{24}-1)$ and must contain elements which can represent any rational number between 10^{-30} and 10^{30} with an accuracy of 1 part in 10^{6}. A printer may represent other rationals as well, but is not required to do so.

Throughout this specification, integer means either a mathematical integer or a Number which is a mathematical integer. A *Cardinal* is an integer in the range 0..*maxCardinal*; the Cardinals are a subset of the Numbers and not a completely distinct type. A Cardinal literal is expressed in the usual decimal notation. Examples: 0, 17.

A Number literal is either a sequence of decimal digits, possibly preceded by a minus sign, or a rational number expressed as a quotient of two such integers, separated by a "/" character. Examples: −2, 17/1, 7/4, 10/72.

2.2.2 Identifiers

An I*dentifier* is a sequence of lower-case letters, digits, and the minus character "−", beginning with a letter. The maximum length of the sequence is *maxIdLength*. An Identifier literal is simply a suitable sequence of characters. Upper-case letters may be included, but are mapped to the corresponding lower-case letters; i.e., case is not distinguished. Examples: *Helvetica*, *old−x*, *z12*.

2.2.3 Marks†

A *Mark* is a distinguished value which can only be pushed on the stack by a MARK operator, and can be removed only by a *matching* UNMARK operator, i.e., one executed in the same context (§2.3.2), or during a mark recovery (§2.4.1). Any other attempt to pop a Mark causes a master error. Marks thus serve to limit the effects of other operators on the stack and as a left bracket for a group of values (§2.4.6), as well as to direct error recovery (§2.4.1).

2.2.4 Vectors†

A *Vector* is a set of values called its *elements*, and some *indexing* information that allows the elements to be named unambiguously. The maximum size of a Vector is *maxVecSize*. The elements of a Vector form a sequence named by Cardinals. Its indexing information is two Cardinals called the *lower bound* l and the *upper bound* u, which are fixed when the Vector is created; l must be less than or equal to $u+1$. There are $u-l+1$ elements in the sequence. The i^{th} element in the Vector is named by the Cardinal $l+i-1$. A Vector is constructed by MAKEVEC or MAKEVECLU.

There are no literals of type Vector.

But there are primitive operators to make vectors (§2.4.3). The encoding has a convenient way to express calls on these primitives with literal arguments (§2.5). Like all values, vectors are constant, i.e., the value of a vector element cannot be changed, except for the frame (§2.3). Shorthand notations for writing vector constructs are given in §2.4.

2.2.5 Bodies and Operators†

An *Operator* is an Interpress program that can be executed. Executing an Operator causes *state transitions*, as described in §2.3.

An operator is either *primitive*, or *composed*. A primitive operator is an operator built into Interpress. The meaning of each primitive operator (i.e., its state transition function) is explained as part of its definition in this document. The explanation is given informally, in English and pictures, or sometimes as a sequence of other primitive operators, for which the one being defined is a convenient abbreviation. A primitive Operator literal is a sequence of letters in small capitals, e.g., MARK, DO, MAKEVEC. The value of the literal is the primitive operator with that name, as defined in this document.

Certain *special* primitive operators are defined in order to make it easy to define the meaning of other primitives, and cannot actually be written in Interpress masters. These operators have an * prefixed to their names in this document; they do not have any corresponding literals.

A composed operator consists of a Body and a Vector called its *initial frame*; its meaning (i.e., how its transition function is determined by the Body and the initial frame) is explained in §2.4.2. Composed operators can be constructed through the execution of the master or obtained from the environment by the primitives FINDOPERATOR, FINDDECOMPRESSOR, FINDCOLOROPERATOR, and FINDCOLORMODELOPERATOR. There are no composed operator literals.

Composed operators which are constructed by the master are constructed by the MAKESIMPLECO operator (§2.4.5). A composed operator is analogous to an Algol or Pascal procedure: the body is the body of the procedure and the initial frame is the local variables.

A *Body* is a sequence of literal values; the maximum length of the sequence is *maxBodyLength*. A Body literal is a sequence of values of the literals bracketed by { and }. The value of the literal is the sequence of values of the literals within the { } brackets. A Body value can be used *only* as an operand of an *immediately* following *body operator* (MAKESIMPLECO, DOSAVESIMPLEBODY, IF, IFELSE, IFCOPY, CORRECT) or in the skeleton (see §3.1). It is a master error to execute any other literal with a Body on top of the stack.

This restriction permits the encoding to facilitate sequential processing by putting the operator before the body in all cases. This usually allows the body to be executed as it is read unless the operator is MAKESIMPLECO. If the operator appeared second, the body would have to be stored away until the operator came into view.

Bodies are the *only* mechanism used in Interpress for grouping parts of a master into larger executable units. Thus:

- Conditional execution is provided by the IF, IFELSE, and IFCOPY operators, which take a Body and a Cardinal as arguments and execute the Body if the Cardinal is non-zero. For example, the following fragment executes a conditional body if the value of the second frame variable is greater than 3: <2 FGET 3 GT {conditional body} IF>.
- A line of symbols whose positions may require slight corrections (e.g., to accommodate small differences between the font definitions available to the creator and the printer; see §4.10) is generated by an execution of the CORRECT operator, whose argument is a single Body which is executed twice, first to compute the correction parameters, and then to produce the output image for the line.
- The entire master is made of Bodies, held together by a non-executable *skeleton* structure (§3.1).

In all these cases, the isolation between the operator and its caller makes it easy to compose the master in a modular fashion.

In principle, it would be sufficient to use bodies only as operands to MAKESIMPLECO. The composed operators thus generated could then be executed immediately. This would be likely to cause poor performance, however, unless the implementation recognized the important special cases. To reduce the need for cleverness in the implementation, Interpress requires a body as the main operand of the other primitives just enumerated; these primitives convert the body into a composed operator which is then executed once, twice, or conditionally. An existing composed operator *o* can be used as the operand of a body operator by applying DO and enclosing it in brackets, i.e., {*o* DO}.

Examples of bodies:

```
{-- draw a solid box with size given by the top two stack values, width and height --
    TRANS
    0 0 4 2 ROLL
    MASKRECTANGLE
}

{-- draw a hollow box with size given by the top two stack values, width and height --
 -- save the box height in frame element 1, the width in element 0 --
    1 FSET 0 FSET
    TRANS 0 0 MOVETO
    0 FGET LINETOX 1 FGET LINETOY 0 LINETOX 0 LINETOY
    MASKSTROKECLOSED
}
```

2.3 State

The state of the machine that executes Interpress masters consists of:

- The global variables and the local variables of the *DoMaster* procedure and of all instances of the *DoBlock* procedure in the midst of execution (§3.1),
- The *stack* (§2.3.1),
- The *contexts* of composed operators in the midst of execution (§2.3.2),
- The *imager variables* (§4.2).

In addition, there is information outside the machine, such as the image being constructed. This information, which is called *output*, is of course the reason for the existence of an Interpress master. Unlike the state, however, it cannot affect any future state.

The global variables are alterable only by the execution of the *DoMaster* and *DoBlock* procedures. Executing an operator causes changes in the state of the machine, or in the output, or both. These changes may (and generally do) depend on the current state. Thus the meaning of an operator can be completely defined by two *transition functions*:

- a *state transition function*, which maps a state of the Interpress machine to a new state of the machine;
- an *output transition function*, which maps a pair: (state of the Interpress machine, output) to a new output.

Note that the output does not affect the state of the Interpress machine. In other words, output is write-only; it cannot be read back to influence later execution.

2.3.1 The stack

The stack is a sequence of values on which the usual push and pop operations are defined; the maximum length of this sequence is *maxStackLength*. It is used primarily to pass arguments to an operator and to obtain results in return. The stack is the *only* general way to return values from an operator. Execution modifies the stack as described in §2.4 and the primitive operator definitions.

2.3.2 Frames and contexts†

A composed operator is constructed from a Body and a Vector which is called its *initial frame*. Each time the operator is executed, a *context* is created to represent this execution. The context contains a return link to the calling operator's context (not directly accessible to the master) and a vector called the *frame*, which is initialized to the value of the operator's initial frame. During execution, elements in the frame can be changed with the FSET operator and read with the FGET operator. The frame itself is not shared, and can be touched only by the FGET and FSET operators executed in its context. After the composed operator is finished, the frame and context are discarded.

Thus an operator can have local variables and can also access (through its initial frame) some global values available when it is defined. Changes to local variables cannot affect the state after execution of the operator is complete, however, except by values returned on the stack. The effect is like local and global variables in Algol or Pascal, except that the global variables are all read-only. Because results can be returned on the stack, an operator can return an arbitrary amount of information as explicit results (contrast the restriction to a single scalar function result in Algol or Pascal). On the other hand, it cannot cause side effects by changing global variables or variable parameters (unless it calls an imager operator, which changes an imager variable).

2.4 Operators

The meaning of an operator is completely defined by its state transition function and its output transistion function. In the definitions of operators below and in the rest of this standard, the state transition function of an operator *Op* is defined by text which begins:

$<a_1: T_1>...<a_n: T_n>$ *Op* → $<r_1: U_1>...<r_m: U_m>$
where ...

The symbols a_1 ... a_n represent argument values, r_1 ... r_m result values, and T_1 ... T_n and U_1 ... U_m their types. If a result r_i is the same as the argument a_j, then the type U_i is the same as T_j and may be omitted.

This text means that in an error-free execution of the operator:

- First, *n argument* values are popped off the stack and given the names a_n (for the first value popped) through a_1 (for the last value popped) for use in the definition. If no arguments are popped, < > appears to the left of *Op*.
- Then some function of the a_i (specified by text after the **where**) is used to compute *m result* values r_i with the indicated types.
- Finally, the results are pushed onto the stack (r_1 first, r_m last). If no results are pushed, < > appears to the right of the → symbol.

The <*name: type*> sequences give a picture of the top of the stack before and after the operator is executed. Note that all the values mentioned on the left of the → are always popped, and hence there is a master error if any turns out to be a mark. This is true even for operators like COPY which push their arguments back again.

The description following the **where** is sometimes informal English and sometimes an Interpress program. The latter means that executing the primitive being defined has the same effect as executing the defining program. An argument name appearing in the program means that the corresponding value is pushed onto the stack at that point; the defining program thus begins executing after the arguments have been popped, but it is responsible for pushing the results. Sometimes these programs are not true Interpress, but use a pseudo-Pascal instead, in which Pascal variables are treated as elements of an Interpress frame. These programs often use familiar Pascal control constructs such as **if then else**, **for**, and **while**, which do not exist in true Interpress.

It is often convenient to specify a value by giving an Interpress program which computes it and leaves it on top of the stack. When such a program appears in text it is enclosed in < > brackets. Thus <3 4 ADD> stands for the value 7.

Shorthand notation

The following shorthand notations are used in the text. Each is simply a more readable way of writing an Interpress construct. These shorthands are not part of the encoding. (The < > and [] with literal numbers can also be considered shorthands for the string and large vector encoding notations; see §2.5.3.)

[x_0, ..., x_{k-1}] stands for x_0 ... x_{k-1} k MAKEVEC, where the x_i are of type Any. Hence [] stands for 0 MAKEVEC. This is simply a convenient way of writing certain uses of MAKEVEC; elements of the vector are separated by commas. The brackets and commas are not part of Interpress.

<*sequence of characters*> stands for n_0, n_1 ... n_{k-1} k MAKEVEC, where n_i is a Cardinal that indexes the corresponding character in the font used by SHOW. The brackets are not part of Interpress.

$n_0/n_1/.../n_{k-1}$ stands for n_0, n_1, ..., n_{k-1} k MAKEVEC, where the n_i are Identifiers. This notation is used for structured names (§3.2). The / character is not part of Interpress.

2.4.1 Errors

If the value named a_i doesn't have the type T_i, there is a *master error*; note that during execution of the master there are always marks on the stack which prevent underflow (see §3.1). The definition may specify further conditions which must be satisfied; if the current state does not satisfy these conditions there is also a master error. In case of a master error, unless the operator definition specifies otherwise, there is a *mark recovery*.

A mark recovery also occurs whenever any attempt is made to pop a mark except by a matching UNMARK or COUNT (§2.4.6); when this happens, the mark remains on the stack.

On a mark recovery,

a) the stack is popped until a mark is on top;

b) if the context that placed the mark on the stack no longer exists, the mark is popped and there is another mark recovery;

c) otherwise, composed operators in execution are exited until the one which placed the mark is executing; and

d) literals are skipped in this operator until an UNMARK0 literal is found.

The UNMARK0 found in step d) is then executed, thus popping the mark from the stack, and execution proceeds from this point. Note that the rules for executing a master given in §3.1 insure that step d) will eventually succeed. In addition, master errors are logged as specified in §5.3.

Marks thus serve two major purposes: to protect the stack from damage by an operator, and to indicate possible error recovery points.

2.4.2 Composed operators†

Composed operators are of two types: those obtained by FINDOPERATOR and those constructed by the master. A composed operator obtained by FINDOPERATOR is executed by applying its state transition function to the current state, yielding a new state. A composed operator constructed by the master is executed by executing the literals of its Body in order. Executing a Number, Identifier, or Body literal pushes its value onto the stack. Executing a primitive operator literal executes the corresponding primitive operator, i.e., applies its state transition function to the current state, yielding a new state.

When execution of the composed operator begins, the frame is initialized to the operator's initial frame, as discussed in §2.3.2; its contents may then be changed by FSET. These changes have no effect on the frame of any other context; each context has its own frame.

Not only does execution of the operator not affect the frame afterwards, but the effect of the operator does not depend on the frame when it is invoked. The effect depends only on its initial frame (established when it was defined) and the stack.

2.4.3 Vector operators†

<v: Vector> <j: Cardinal> GET → <x: Any>
where x is the value of the element of v named by j. A master error occurs unless $l \leq j \leq u$, where l is the lower bound of v and u is the upper bound.

<x_1: Any>...<x_n: Any> <l: Cardinal> <u: Cardinal> MAKEVECLU → <v: Vector>
where v is a vector with lower bound l and upper bound u. Let $n = u - l + 1$. After u and l are popped off the stack, n additional values are popped; call them x_n, ... , x_1, where x_n is the first value popped and x_1 is the last value popped. The elements of v have the values x_1, ..., x_n; i.e., <v $l+i-1$ GET> = x_i. A master error occurs unless $0 \leq n \leq maxVecSize$.

<x_1: Any>...<x_n: Any> <n: Cardinal> MAKEVEC → <v: Vector>
where v is a vector with lower bound 0 and upper bound n-1. The elements of v have the values x_1, ..., x_n; i.e., <v $i-1$ GET> = x_i. A master error occurs unless $n \leq maxVecSize$. §2.4 describes a notation for writing calls of MAKEVEC in examples.

<v: Vector> SHAPE → <l: Cardinal> <n: Cardinal>
where l is the lower bound of v and the upper bound is $u = l + n - 1$.

A *property vector* is a vector formatted according to a convention that elements with indices l, $l+2$, $l+4$, and so on are *property names* and elements with indices $l+1$, $l+3$, $l+5$, and so on are corresponding *values*. For example, in the vector [*widthX*, 14, *widthY*, 21], the property named *widthX* has value 14 and the property *widthY* has value 21. The intent is that Cardinals, Identifiers, and Vectors of Identifiers be used as property names; other values are permitted, but may not be found by GETPROP. The following operators are defined for all vectors but are intended for use with property vectors:

<v: Vector> <*propName*: Any> GETPROP → <*value*: Any> <1: Cardinal>
or → <0: Cardinal>
where v is searched to find the least i such that $l \leq i \leq u$ (l is v's lower bound and u is its upper bound), $(i-l)$ **mod** 2 = 0, and <v i GET *propName* *EQN> = 1. *EQN compares Identifiers, Numbers, and Vectors for equality; it is defined precisely in §2.4.8. If no match is found, GETPROP returns 0 on the stack. If a match is found, <v i 1 ADD GET 1> is executed to place on the stack the property's value and the Cardinal 1, which indicates that a match has been found. A master error occurs if $(u-l+1)$ **mod** $2 \neq 0$.

<v: Vector> <*propName*: Any> GETP → <*value*: Any>
where v is searched to find the value of *propName* as with GETPROP, except that a match for *propName* must be found. When a match is found, <v i 1 ADD GET> is executed to place the property's value on the stack. A master error occurs if $(u-l+1)$ **mod** $2 \neq 0$ or if no match is found.

<v_1: Vector> <v_2: Vector> MERGEPROP → <v_3: Vector>
where the property vector v_3 is created by merging properties and values from v_1 and v_2, so that values in v_2 take priority over values in v_1. More precisely, v_3 is formed so that, for any n, <v_3 n GETPROP> is equivalent to <v_2 n GETPROP DUP NOT { POP v_1 n GETPROP } IF>. A master error occurs if either the number of elements in v_1 **mod** $2 \neq 0$ or the number of elements v_2 **mod** $2 \neq 0$.

Note that this definition does not fully specify the number or order of elements in v_3. Consequently, the results of <v_3 SHAPE> and <v_3 n GET> are implementation dependent. The definition does, however, fully define the behavior of v_3 when accessed with GETP or GETPROP.

2.4.4 Frame operators†

<*j*: Cardinal> FGET → <*x*: Any>
where *x* is the current value of the j^{th} element of the frame. A master error occurs unless $j < topFrameSize$. The value of *topFrameSize* may be limited (§5.1.1).

<*x*: Any> <*j*: Cardinal> FSET → < >
where the value of the frame element named by *j*, becomes *x*. A master error occurs unless $j < topFrameSize$.

2.4.5 Operator operators†

<*b*: Body> MAKESIMPLECO → <*o*: Operator>
where *o* is a composed operator which has Body *b* and initial frame equal to the value of the frame when MAKESIMPLECO is executed.

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

This is the only way to execute a composed operator (other primitives which do this are defined in terms of DO; they are DOSAVE, DOSAVEALL, DOSAVESIMPLEBODY, the three IF operators, and CORRECT).

<*o*: Operator> DOSAVE → -- the effect on the stack depends on *o* --
where the effect is equivalent to executing the operator *o* with DO, and then restoring all the non-persistent imager variables to their values just before the DOSAVE.

<*o*: Operator> DOSAVEALL → -- the effect on the stack depends on *o* --
where the effect is equivalent to executing the operator *o* with DO, and then restoring all imager variables to their values just before the DOSAVEALL.

An additional operator executes a Body but saves the non-persistent imager variables.

<*b*: Body> DOSAVESIMPLEBODY → -- the effect on the stack depends on *b* --
where the effect is *b* MAKESIMPLECO DOSAVE.

The FINDOPERATOR primitive obtains a composed operator from the printer environment. The effect of executing the operator must be consistent with the model of operator execution (§2.3) and the imaging model (§4.1).

<*v*: Vector> FINDOPERATOR → <*o*: Operator>
where *v* is a Vector of Identifiers, which is the universal name of the operator *o*.

The effect of executing such an operator may be anything which could be accomplished by a sequence of Interpress literals, including a sequence of literals which was constructed at printing time (such as a representation of the current date and time). Execution of the operator may also produce side effects which are outside of the state of the Interpress machine and its output, such as invoking the scanning of a document in a recirculating document handler and storing the resultant scanned image in a known file.

2.4.6 Stack operators†

<*x*: Any> POP → < >
where the top element of the stack is removed with no other effects.

$<x_1$: Any$>$...$<x_{depth}$: Any$>$ $<$*depth*: Cardinal$>$ COPY → $<x_1>$...$<x_{depth}>$ $<x_1>$...$<x_{depth}>$

where the *depth* values are pushed, leaving the stack in the same state as after *depth* is popped, and then the same *depth* values are pushed again in the same order.

$<x$: Any$>$ DUP → $<x>$ $<x>$

where the effect is x 1 COPY; i.e., the top element of the stack is duplicated with no other effects.

$<x_1$: Any$>$...$<x_{depth}$: Any$>$ $<$*depth*: Cardinal$>$ $<$ *moveFirst*: Cardinal$>$ ROLL → $<x_{moveFirst+1}>$...$<x_{depth}>$ $< x_1>$...$<x_{moveFirst}>$

where $moveFirst \leq depth$, the first *moveFirst* of the *depth* argument values become the last *moveFirst* of the *depth* result values, and order is otherwise preserved.

$<x$: Any$>$ $<y$: Any$>$ EXCH → $<y>$ $<x>$

where the effect is x y 2 1 ROLL; i.e., the top two elements on the stack are exchanged.

$<x_1$: Any$>$...$<x_n$: Any$>$ $<n$: Cardinal$>$ MARK → $<m$: Mark$>$ $<x_1>$...$<x_n>$

where m is a Mark unique to the current context. Only an execution of k UNMARK in the same context with k values above m on the stack can remove m from the stack without an error. The x_i values are unaffected.

$<m$: Mark$>$ $<x_1$: Any$>$...$<x_n$: Any$>$ $<n$: Cardinal$>$ UNMARK → $<x_1>$...$<x_n>$

where m is a matching Mark, i.e., one pushed by a MARK in the same context. The only effect is to remove m from the stack; the x_i values above it are unaffected.

For example, the following sequence executes an operator o which is supposed to take two arguments and return three results; it ensures that o does not pop additional values from the stack and that it returns exactly three results: 2 MARK o DO 3 UNMARK.

$<m$: Mark$>$ UNMARK0 → $<>$

where the effect is 0 UNMARK. UNMARK0 also serves as a stopping point for a mark recovery (§2.4.1).

$<m$: Mark$>$ $<x_1$: Any$>$...$<x_n$: Any$>$ COUNT → $<m$: Mark$>$ $<x_1>$...$<x_n>$ $<n$: Cardinal$>$

where m is a matching Mark, i.e., one pushed by a MARK in the same context. The only effect is to count the number of values above m on the stack and push this count. The values and m are unaffected.

$<>$ NOP → $<>$

where execution of this operator has no effect on the state or output.

$<$*message*: Vector of Cardinal$>$ $<$*code*: Cardinal$>$ ERROR → $<>$

where the Interpress interpreter records that an error has occurred and executes an error handling procedure appropriate to the severity of the error. The value of *code* determines the kind of error (§5.3):

0	master error
10	master warning
50	appearance error
60	appearance warning
100	comment

The vector *message* specifies an explanatory message by giving the character codes for characters in the message using the ISO 646 7-bit Code Character Set for Information

Processing Interchange. The message is handled similarly to error messages for other errors of like severity.

For example, <[47, 111, 112, 115] 100 ERROR> would record the comment "Oops"

2.4.7 Control operators†

<*i*: Cardinal> <*b*: Body> IF → -- the effect on the stack depends on *i* and *b* --
where the effect is *b* MAKESIMPLECO DO if $i \neq 0$, and nothing otherwise.

<*i*: Cardinal> <*b*: Body> IFELSE → -- the effect on the stack depends on *i* and *b* --
where the effect is *i b* IF *i* 0 EQ; i.e., it is the same as the effect of IF, followed by pushing 1 if $i = 0$ and 0 otherwise.

Note that *i* is not on the stack when the body is executed. The effect of "**if** *i* **then** B1 **else** B2" is obtained with "*i* B1 IFELSE B2 IF". The effect of "**if** *i*1 **then** B1 **else if** *i*2 **then** B2 **else** B3" is obtained with "*i*1 B1 IFELSE { *i*2 B2 IFELSE B3 IF } IF". The funny way IFELSE works allows each operator to have exactly one body operand. Because IF executes a Body conditionally (by turning it into an operator), any changes the Body makes to its frame are discarded when the execution is complete.

It is often valuable to print several copies from a master which differ in minor details; for example, each copy might be addressed to a different recipient on the first page. It is important to ensure that these variations do not require the entire master to be reprocessed for each copy. The IFCOPY operator serves this purpose.

<> *COPYNUMBERANDNAME → <*copyNumber*: Number> <*copyName*: Identifier>
where the values returned are the copy number and copy name, obtained in a manner described in §3.1. This operator cannot be called directly from the master.

<*testCopy*: Operator> <*b*: Body> IFCOPY → <>
where *testCopy* is called with the copy number and copy name on the stack, and *b* is executed unless *testCopy* returns 0. The *testCopy* operator can execute only BASE operators (§3.1.1). Precisely, the effect is:

```
<0 MARK
    *COPYNUMBERANDNAME testCopy DOSAVEALL
    {0 MARK b MAKESIMPLECO DOSAVEALL UNMARK0} IF
UNMARK0>.
```

In other words, the *testCopy* operator takes two arguments, and must return a single Cardinal; if this is non-zero, the Body *b* is executed. Both executions are done with DOSAVEALL; this and the MARK/UNMARK pairs ensure that there are no side effects. Thus the net result is that *testCopy* decides whether or not to print the output produced by *b*. A different decision can be made for each copy, but either nothing or the same output is produced each time.

2.4.8 Test operators†

<*a*: Any> <*b*: Any> EQ → <*c*: Cardinal>
where $c = 1$ if *a* and *b* are both Numbers or both Identifiers and $a = b$, $c = 0$ otherwise.

<*a*: Any> <*b*: Any> *EQN → <*c*: Cardinal>
where $c = 1$ if <*a b* EQ> is 1 or if *a* and *b* are both Vectors with the same shape and corresponding elements are EQ; $c = 0$ otherwise. More precisely, *EQN is equivalent to:

```
i,n,l: Cardinal;
if a b EQ then { 1 } else {
if a TYPE 3 EQ b TYPE 3 EQ AND then {
    a SHAPE n FSET l FSET
    if b SHAPE n FGET EQ EXCH l FGET EQ AND then {
```

```
            1 c FSET
            for i := l to l+n−1 do {
                if a i FGET GET b i FGET GET EQ NOT then { 0 c FSET }
            } c FGET
        } else { 0 }
    } else { 0 } }
```

<a: Number> <b: Number> GT → <c: Cardinal>
where $c=1$ if $a > b$, $c=0$ otherwise.

<a: Number> <b: Number> GE → <c: Cardinal>
where $c=1$ if $a \geq b$, $c=0$ otherwise.

<a: Cardinal> <b: Cardinal> AND → <c: Cardinal>
where $c=1$ if $a \neq 0$ and $b \neq 0$, $c=0$ otherwise.

<a: Cardinal> <b: Cardinal> OR → <c: Cardinal>
where $c=1$ if $a \neq 0$ or $b \neq 0$, $c=0$ otherwise.

<b: Cardinal> NOT → <c: Cardinal>
where $c=1$ if $b=0$, $c=0$ otherwise.

<a: Any> TYPE → <c: Cardinal>
where c is the code for the type of a as specified in Appendix B.1.

2.4.9 Arithmetic operators†

<a: Number> <b: Number> ADD → <c: Number>
where $c = a+b$.

<a: Number> <b: Number> SUB → <c: Number>
where $c = a-b$.

<a: Number> NEG → <c: Number>
where $c = -a$.

<a: Number> ABS → <c: Number>
where $c = |a|$.

<a: Number> FLOOR → <c: Number>
where c is the greatest integer $\leq a$.

<a: Number> CEILING → <c: Number>
where c is the least integer $\geq a$.

<a: Number> TRUNC → <c: Number>
where c is the integer part of a. Thus <7/2 TRUNC> = 3 and <−7/2 TRUNC> = −3.

<a: Number> ROUND → <c: Number>
where the effect is a 1/2 ADD FLOOR; i.e., c is the rounded value of a.

<a: Number> <b: Number> MUL → <c: Number>
where $c = a \times b$.

<a: Number> <b: Number> DIV → <c: Number>
where $c = a/b$; $b \neq 0$ is required. Note that this is rational division, *not* integer division. (The Pascal operator **div**, written in lower case and bold face, denotes integer divison, which is distinct from DIV.)

<a: Number> <b: Number> MOD → <c: Number>
where $c = a - b \times \text{FLOOR}(a/b)$; $b \neq 0$ is required. (The Pascal operator **mod**, written in lower case and bold face, is distinct from MOD.)

<a: Number> <b: Number> REM → <c: Number>
where $c = a - b \times \text{TRUNC}(a/b)$; $b \neq 0$ is required.

2.5 The Xerox encoding

This section gives the rules for encoding an Interpress master. The principal job of an encoding is to specify how every legal sequence of Interpress literals can be represented concretely by a collection of bits that may be stored or transmitted. In addition, the encoding introduces some shorthand notations that can be used in place of more bulky notations for sequences of Interpress literals; these are termed *encoding notations* (§2.5.3).

Many computer file systems use a short file-name "extension" that serves to indicate the type of the file. Extensions are sometimes two or three characters long, e.g., LST for listing, BIN for binary, EXE for executable. Programs that create Interpress masters and store them on disk files are urged to use the extension "Interpress", or, if extensions must have fewer than ten characters, "IP".

The master is encoded by a *header* which identifies the encoding, followed by a sequence of *tokens*. Each token corresponds to:

A single Interpress literal (not a Body). Each such literal can be encoded by a single token.

One of the symbols BEGIN, END, CONTENTINSTRUCTIONS, "{", or "}".

An encoding notation, which stands for some sequence of Interpress literals.

The tokens appear in the same order as the corresponding literals or symbols, except that a body operator token precedes its body.

The tokens are of different sizes; each one is a sequence of bytes. A byte is an integer in the range 0..255 inclusive, and is represented by eight bits. The encoding is defined below by giving Pascal-like programs that invoke the function *AppendByte*(*n*) to append a byte with value *n* to the sequence being created to encode the master; the programs use infinite-precision integer arithmetic. It is also convenient to draw diagrams of the encoding. In these diagrams, bytes are shown juxtaposed so as to be read from left to right, i.e., the byte at the far left appears first in the sequence, followed by the byte to its right, etc. The diagram of a single byte shows its 8 bits, with the most significant bit (corresponding to 2^7) at the left and the least significant (corresponding to 2^0) at the right.

The first bytes of a master in the Xerox encoding are the header that identifies the encoding and version number. These bytes are an encoding of the string "`Interpress/Xerox/3.0`□" using character codes from the ISO 646 7-bit Coded Character Set for Information Processing Interchange. The symbol "□" is used in this section to represent the space character, which has code 32 in ISO 646. It is the space character that terminates the header. The header can be created by a sequence of calls to *AppendByte*:

AppendByte(73); -- I -- *AppendByte*(110); -- n -- *AppendByte*(116); -- t --
AppendByte(101); -- e -- *AppendByte*(114); -- r -- ....

AppendByte(120); -- x -- *AppendByte*(47); -- / -- *AppendByte*(51); -- 3 --
AppendByte(46); -- . -- *AppendByte*(48); -- 0 -- *AppendByte*(32); -- □ --

Following the header come encodings of the parts of the skeleton (§3.1).

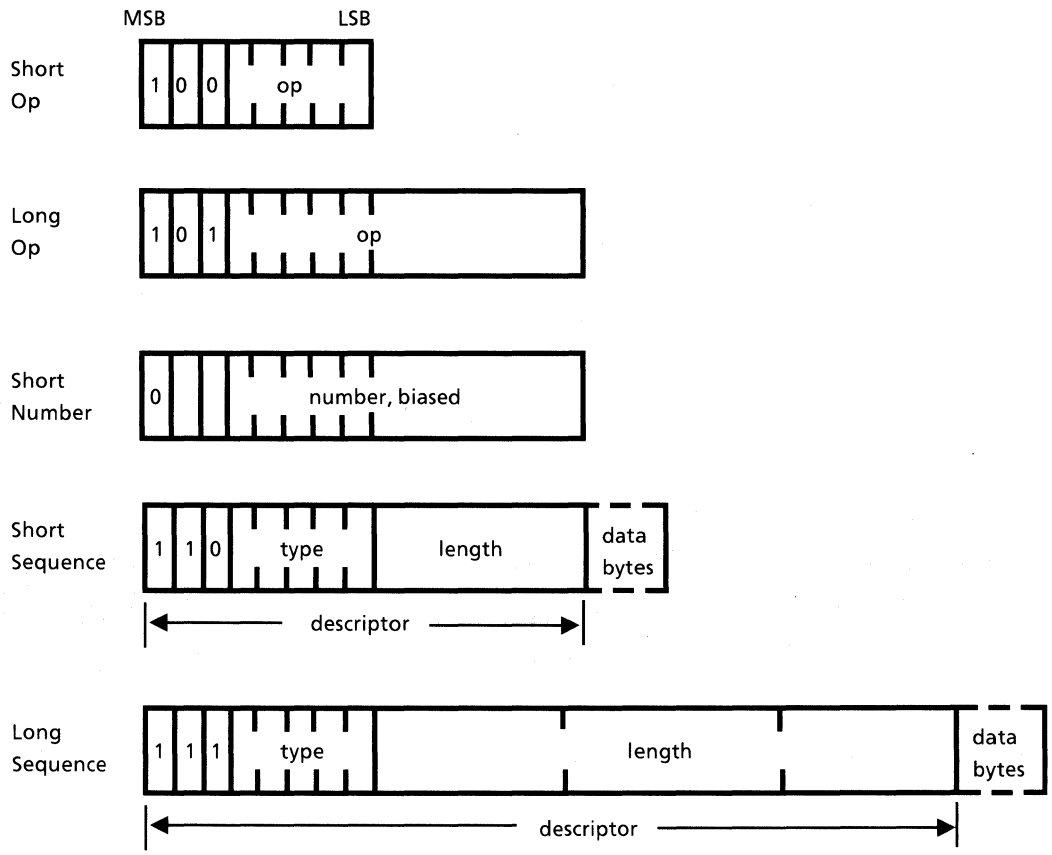


Figure 2.1 **Token formats**

2.5.1 Token formats

Each token uses one of five formats: Short Op, Long Op, Short Number, Short Sequence, and Long Sequence (see Figure 2.1). The token formats are described in this section, and the rules for encoding literals in tokens are described in §2.5.2.

Each primitive operator or symbol is assigned an integer in the range 0..8191 called its *encoding-value*, and is represented in the master by a two-byte Long Op token, or optionally by a one-byte Short Op token if its encoding-value is less than 32; the details are given below.

An integer in the range −4000..28767 may be represented by a two-byte Short Number token as described below.

Everything else is represented by variable-length Short Sequence and Long Sequence tokens. These begin with a two or four byte *descriptor* which includes a *length* field that gives the number of *data bytes* used to represent the value, and a *type* field that indicates

what kind of literal or encoding notation the data bytes represent. The *length* field gives the number of data bytes; it does *not* count the bytes that are part of the descriptor.

The following *AppendSequenceDescriptor* procedure generates the descriptor for a sequence of length *length* and type *seqType*:

procedure *ExtractByte*(*n*, *byte*: integer): integer;
-- *Extract from a positive integer* n *the* byte*th byte required to represent it, counting low-order byte as 0.* --
begin for $i := 1$ **to** *byte* **do** $num := num$ **div** 256; *ExtractByte* := num **mod** 256 **end**;

procedure *AppendInt*(*num*, *length*: integer);
-- *Encode an integer in* $(-256^{length}$ **div** $2)..(256^{length}$ **div** $2-1)$ *in twos-complement* --
-- *using* length *bytes, high-order byte first.* --
begin
if $num<0$ **then** $num := 256^{length}+num$;
if $num<0$ **or** $num \geq 256^{length}$ **then error**;
for $i := 1$ **to** *length* **do** *AppendByte*(*ExtractByte*(*num*, *length* $-i$))
end

procedure *AppendSequenceDescriptor*(*seqType*, *length*: integer);
begin
if *seqType* <0 **or** *seqType* >31 **then error**
else if *length* <0 **or** *length* >16777215 **then error**
else if *length* <256 **then begin**
-- *Short Sequence, with one byte of length* --
AppendByte(192 + *seqType*); *AppendByte*(*length*) **end**
else begin
-- *Long Sequence, with three bytes of length* --
AppendByte(224 + *seqType*); *AppendInt*(*length*, 3) **end**
end

Any sequence token can be *continued* by one or more immediately following sequence tokens (either Short or Long) with the type *sequenceContinued*. A sequence token with $seqType=t$ and $length=l$ followed by a *sequenceContinued* token with $length=m$ is equivalent to a single sequence token with $seqType=t$, $length=l+m$, and data bytes which are the l data bytes of the first token followed by the m data bytes of the second. If there are several consecutive continuations, this merging may be repeated until they have all been merged into the initial non-continuation sequence token.

Continuations make it convenient for a creator to break a long sequence into several shorter pieces, e.g., to fit into its limited buffers. The total length of a continued sequence may exceed 16777215 bytes. A continuation token may have $length=0$.

2.5.2 Literal encodings

Number. A number may be encoded in one of three ways, all of which result in the same Interpress Number value; any of these ways may be used to create a legal master. Since Cardinals are a subset of Numbers and not a totally distinct type, it is not compulsory to encode a Cardinal using one of the encodings which works only for integer values. In general, however, a master will be smaller and will also be interpreted more efficiently if the shortest encoding is chosen. Although the encoding can represent Numbers with great range and precision, a printer is required to represent only a subset of these (§2.2.1).

- If the number is an integer and lies in the range −4000..28767, it may be encoded in a Short Number token, with a bias of 4000. The *AppendInteger* procedure below generates this encoding or the next as appropriate.

- If it is an integer it may be encoded in a sequence token of type *sequenceInteger*, and at least enough data bytes to represent it as a signed two's complement binary integer. The *AppendInteger* procedure below generates this encoding for an integer which cannot be encoded as a Short Number, using a minimum number of data bytes. Note the treatment of negative numbers by *AppendInt* to produce a two's complement encoding.

- A number may be encoded as a *rational*, a quotient of two integers. The two integers are encoded in a sequence token of type *sequenceRational*, numerator first, both using the same number of bytes. The *AppendRational* procedure below generates this encoding, using a minimum number of data bytes.

```
function BytesInInt(n: Number): integer;
    var done: boolean := false; i: integer:= 0;
    begin
    until done do begin
        i := i+1; if –(256^i div 2) ≤ n and n <(256^i div 2) then done := true end;
    BytesInInt := i
    end;

procedure AppendInteger(n: Number);
    const i = BytesInInt(n);
    begin
    if −4000 ≤ n and n ≤ 28767 then AppendInt(n+4000, 2)
    else begin AppendSequenceDescriptor(sequenceInteger, i); AppendInt(n, i) end
    end;

procedure AppendRational(n, d: Number);
    const i = Max(BytesInInt(n), BytesInInt(d));
    begin AppendSequenceDescriptor(sequenceRational, 2*i); AppendInt(n, i);
    AppendInt(d, i) end
```

Identifier: An identifier is encoded using a sequence of character codes that represent the characters of the identifier, which are limited to letters, digits, and "–". Note that case is not distinguished in identifiers; hence a letter may be encoded in either upper or lower case. Each character in the identifier is represented by a single byte whose value is the ISO 646 code for the character. The first character in the identifier appears first in the sequence, then the second, and so on. The sequence of codes is placed in a sequence token of type *sequenceIdentifier*. Thus the following program would encode the identifier *Xerox*:

```
AppendSequenceDescriptor(sequenceIdentifier, 5);
AppendByte(88);     -- X --     AppendByte(101);  -- e --     AppendByte(114);  -- r --
AppendByte(111);    -- o --     AppendByte(120);  -- x --
```

Primitive operator: A primitive operator is encoded by placing its encoding value in a Short Op or Long Op token; the *AppendOp* procedure below generates a suitable Op token from a numeric value. A table giving the numeric code for each operator appears in Appendix B.3. Thus the LINETO operator is encoded by *AppendOp*(23).

The body operators (MAKESIMPLECO, DOSAVESIMPLEBODY, IF, IFELSE, IFCOPY, CORRECT) are encoded slightly differently: they are placed *before* the encoding of the body that is their final argument. Thus each of these operators *must* be followed immediately by a body encoding.

If the encoding value is less than 32, it can be encoded using a Short Op token; otherwise it must use a Long Op token. Thus the following program encodes a primitive:

procedure *AppendOp*(*n*: integer)
begin
if $n<0$ **or** $n>8191$ **then error**
else if $n \leq 31$ **then** *AppendByte*$(128+n)$
else begin *AppendByte*$(160+n$ **div** $256)$; *AppendByte*$(n$ **mod** $256)$ **end**
end

Body: A body literal is encoded in the obvious way; note that it is *preceded* by its body operator if there is one:

The encoding begins with a token which encodes the "{" symbol, generated with *AppendOp*(106).

Then comes the sequence of literals that form the body.

Finally there is a token which encodes the "}" symbol, generated with *AppendOp*(107).

BEGIN, END, and CONTENTINSTRUCTIONS. These symbols, which are part of the skeleton (§3.1), are encoded with Op tokens.

Comment: An arbitrary sequence of bytes may be embedded in a sequence of type *sequenceComment*. An Interpress printer will ignore this token.

2.5.3 Encoding notations

The encoding includes some notations that do not correspond to individual Interpress literals, but rather to sequences of literals.

String: It is often necessary to encode vectors of small Cardinals used to represent *character codes*; these vectors occur especially frequently as arguments to the SHOW operator (§4.4.6). These vectors may be encoded compactly using a sequence token with type *sequenceString*. The data bytes within the token are encoded using a scheme defined in the Xerox Character Code Standard. *It is important to note, however, that the encoding simply defines a run-length encoding of numbers and does not associate characters with numbers.*

Although the Xerox Character Code Standard defines the encoding scheme, it is summarized here. Loosely, the encoding scheme provides a run-length encoding for a sequence of integers that differ mostly in the low-order 8 bits and an extended encoding that represents each integer with two bytes. An escape byte with value 255 is used to switch modes. The scheme can represent only integers i in the range $0 \leq i \leq 65278$ and such that $(i$ **mod** $256) \neq 255$.

We describe first the way the data bytes are decoded sequentially to obtain the vector of integers they represent. Decoding begins in *simple* mode, and proceeds according to the following rules:

1. *Simple*. If the numeric value b of the byte lies in the range $0 \leq b \leq 254$, a Cardinal with value b is added to the vector, and the subsequent bytes are interpreted similarly. If $b=255$, the next byte n is examined to determine whether to enter the *run* or *extended* modes.

2. *Run*, entered if $0 \leq n \leq 254$. If the numeric value b of the byte lies in the range $0 \leq b \leq 254$, a Cardinal with value $n*256+b$ is added to the vector, and the subsequent bytes are interpreted similarly. If $b=255$, the next byte n is examined to determine whether to enter the *run* or *extended* modes.

3. *Extended*, entered if $n=255$. The immediately following byte must be 0. Subsequent bytes are interpreted as follows: the next byte b_1 is examined; if $b_1 \neq 255$, then the next byte b_2 is obtained, and a Cardinal with value

b_1*256 + b_2 is added to the vector. If $b_1 = 255$, the next byte n is examined to determine whether to enter the *run* or *extended* modes.

The decoding scheme is illustrated in the following examples:

1. *Simple* only.
   Data bytes: 65, 83, 67, 73, 73, 32, 98, 97, 115, 101, 100
   Vector of integers: [65, 83, 67, 73, 73, 32, 98, 97, 115, 101, 100]

2. *Simple* and *run*.
   Data bytes: 102, 111, 255, 239, 48, 255, 0, 111, 116, 110, 111, 116, 101
   Vector of integers: 102, 111, 61232, 111, 116, 110, 111, 116, 101

3. *Simple* and *extended*.
   Data bytes: 255, 255, 0, 1, 97, 0, 32, 33, 98, 4, 32, 38, 97
   Vector of integers: 353, 32, 8546, 1056, 9825

The *DecodeString* procedure below makes the decoding descriptions precise. The procedure is provided with an argument *data*, which are the data bytes of the token, and an argument *length*, which is the number of data bytes. The procedure *AddInt* is called to add an integer to a vector result that is built up.

```
procedure DecodeString(data: Vector; length: Cardinal);
    type mode = {run, escape, escape2, extended, extended2};
    var i, b, offset: Cardinal; state: mode;
    begin
    offset := 0; state := run;
    for i := 0 to length–1 do begin
        b := data [i];
        case state of
    run:        if b ≠ 255 then AddInt(offset*256 + b) else state := escape;
    escape:     if b ≠ 255 then begin offset := b; state := run end else state := escape2;
    escape2:    if b = 0 then state := extended else error;
    extended:   if b ≠ 255 then begin offset := b; state := extended2 end else state := escape;
    exended2:   begin AddInt (offset*256 + b); state := extended end
                end;
        end;
    if not (state = run or state = extended) then error
    end
```

The *AppendString* procedure shows how a vector might be encoded using only the *run* mode. It performs two passes so that it can compute the total length of the encoding, including escape bytes, during the first pass, while actually constructing the encoding during the second pass.

```
procedure AppendString (v: Vector; numElements: Cardinal);
    var byteCount: Cardinal := 0; offset: Cardinal;
    begin
    for pass := 1 to 2 do begin
        if pass = 2 then AppendSequenceDescriptor(sequenceString, byteCount);
        offset := 0;
        for i := 0 to numElements–1 do begin
            if v [i] < 0 or v [i] > 65278 or v [i] mod 256 = 255 then error;
            if v [i] div 256 ≠ offset then begin
                offset := v [i] div 256;
                if pass = 1 then byteCount := byteCount + 2
                else begin AppendByte(255); AppendByte(offset) end
                end;
            if pass = 1 then byteCount := byteCount + 1 else AppendByte(v [i] mod 256)

            end
        end
    end
```

Large vectors: Image data is often recorded as a large vector of compressed or packed data. For this reason, it is convenient to have a compact representation for large vectors of integers. Sequence type *sequenceLargeVector* is a sequence of bytes that is formed into a vector of integers. The first data byte is equal to the number of bytes which represent each number; call this b. The remaining data bytes are grouped into b byte parts, and each part is a two's-complement representation of an integer. The length of the vector is $(length - 1)/b$; it is required that $(length - 1)$ **mod** $b = 0$. Thus the first b bytes after the initial byte represent a number that will become the vector element with index 0, the next b bytes represent the vector element with index 1, and so on, up to the last b bytes, which represent the vector element with index $(length - 1)/b - 1$.

The following *AppendLargeVector* procedure generates this encoding, using a minimum number of data bytes.

```
procedure AppendLargeVector(v: Vector, numElements: Cardinal);
    var b: integer := 0;
    begin
    for i := 0 to numElements-1 do b := Max(b, BytesInInt(v[i]));
    AppendSequenceDescriptor(sequenceLargeVector, b*numElements+1);
    AppendByte(b);
    for i := 0 to numElements-1 do AppendInt(v[i], b)
    end
```

Pixel vectors: There are several types of sequence tokens which abbreviate a large vector, a call of FINDDECOMPRESSOR, and application of the resulting operator.

A sequence token with type *sequencePackedPixelVector* is equivalent to pushing onto the stack a Vector of Cardinals *v* formed from the data bytes and executing <[*Xerox*, *packed*] FINDDECOMPRESSOR DO>.

A sequence token with type *sequenceCCITT-4PixelVector* is equivalent to pushing onto the stack a Vector of Cardinals *v* formed from the data bytes and executing <[*Xerox*, *CCITT-4*] FINDDECOMPRESSOR DO>.

A sequence token with type *sequenceCompressedPixelVector* is equivalent to pushing onto the stack a Vector of Cardinals *v* formed from the data bytes and executing <[*Xerox*, *compressed*] FINDDECOMPRESSOR DO>.

A sequence token with type *sequenceAdaptivePixelVector* is equivalent to pushing onto the stack a Vector of Cardinals *v* formed from the data bytes and executing <[*Xerox*, *adaptive*] FINDDECOMPRESSOR DO>.

These sequence types require that the printer obtain the specified decompression operator from the environment.

In all cases, the vector *v* is obtained from the data bytes in the encoding by using two bytes to represent each integer. More precisely, if *v* has *numElements* elements, it is encoded with *AppendSequenceDescriptor*(*seqType*, 2**numElements*); **for** i := 0 **to** *numElements*−1 **do** *AppendInt*(*v*[*i*], 2), where *seqType* is one of *sequencePackedPixelVector*, *sequenceCCITT-4PixelVector*, *sequenceCompressedPixelVector*, or *sequenceCompressedPixelVector*.

Inserting from a file: There are two encoding-notations which take a file name as a parameter and whose effect is to replace the sequence token with a sequence of tokens from the named file. Sequence types *sequenceInsertMaster* and *sequenceInsertFile* contain data bytes which are interpreted as the name of a file in the syntax and encoding used in the printer's environment. They differ in where they can occur in the master and in the way in which the tokens are extracted from the named file. File insertions may nest to a depth of *maxFileNesting*.

A sequence token with type *sequenceInsertMaster* references a file which contains a single complete Interpress master. The *sequenceInsertMaster* token is replaced by the tokens of the master's top level block (§3.1), including the enclosing BEGIN and END tokens.

A master error occurs if a *sequenceInsertMaster* token appears within a body.

A sequence token with type *sequenceInsertFile* references a file which contains a sequence of literals or some representation of the effect of executing a sequence of literals. The effect of the *sequenceInsertFile* token is the insertion of this sequence of literals into the master in place of the *sequenceInsertFile* token. A *sequenceInsertFile* token may only appear within a body; a master error occurs if it appears elsewhere. There is one type of file, called *Interpress fragment*, from which the sequence of literals is extracted in a standard manner. Files of other types may be referenced, but the method of replacing the *sequenceInsertFile* token by literals from the named file is not specified by this standard.

An Interpress fragment consists of a standard header followed by a sequence of tokens in the Xerox encoding. The header encodes the string "`Interpress/Xerox/a.b/`*filetype/n.m*□" using character codes from the ISO 646 7-bit Coded Character Set for Information Processing Exchange, where "`a.b`" is a valid Interpress version number, "*filetype/n.m*" is a string of characters describing the type of the Interpress fragment contained in this file and "□" is the space character. The method of replacing the *sequenceInsertFile* token with literals from the file is to skip any tokens preceding the first BEGIN token, and to insert the sequence of tokens following the BEGIN token up to but not including the next END token. A master error occurs if either the BEGIN token or the END token is missing.

One example of an Interpress fragment is the Image File described in the Xerox Raster Encoding Standard, XNSS 178506. The header of such a file might be "Interpress/Xerox/3.0/RasterEncoding/1.0□".

The requirement that the replacement tokens represent a sequence of literals ensures that "{" and "}" tokens are balanced. The effect of a *sequenceInsertFile* token is therefore confined to the body in which the token occurs.

The standard does not specify how the printer determines the type of the named file or the replacement method. In fact, the named file need not actually contain tokens which directly represent a sequence of literals: it might instead contain some representation compiled for a particular printer, as long as the net effect is the same as including some sequence of literals in the master.

It is recommended that all files which use a private encoding, and are referenced by the *sequenceInsertFile* encoding, contain an identifying header. The header should be composed of ISO 646 character codes and should follow the form "Interpress/*universalID*/a.b/ *filetype/n.m#*" where *universalID* is a universal identifier, "a.b" is a valid Interpress version number, and "*filetype/n.m*"is a string of characters describing the type of the private encoding.

2.5.4 Code assignments

All the encoding values except the ones for primitive operators are summarized below; encoding values for primitives are given in Appendix B.3.

Table 2.1 **Encoding values for non-primitives**

Name	Value (decimal)
BEGIN	102
END	103
CONTENTINSTRUCTIONS	105
"{"	106
"}"	107

Table 2.2 **Values for sequence types**

Name	Value (decimal)
sequenceAdaptivePixelVector	12
sequenceCCITT-4PixelVector	13
sequenceComment	6
sequenceCompressedPixelVector	10
sequenceContinued	7
sequenceIdentifier	5
sequenceInsertFile	11
sequenceInsertMaster	3
sequenceInteger	2
sequenceLargeVector	8
sequencePackedPixelVector	9
sequenceRational	4
sequenceString	1

3. GLOBAL STRUCTURE AND EXTERNAL INTERFACE

This chapter describes how an Interpress master is constructed from a sequence of bodies called its *skeleton*. It also explains the mechanisms for interaction with the printer's environment.

3.1 The skeleton

A master has a hierarchical structure. At the top is a body called the *instructions* body and a single *block*. A block consists of a *preamble* body and a sequence of *content* nodes enclosed between BEGIN and END tokens. Each *node* is either another block or simply a body; either may optionally be preceded by a *content instructions* body. Within bodies, the rules of the base language prevail. The collection of nodes and their hierarchical arrangement is called the *skeleton* of the master. The way in which the parts of the skeleton are executed to determine the effects of the master is first outlined and then described in detail.

If an instructions body is present, it is executed first, with the *external instructions* vector on the stack. The external instructions vector is a property vector that encodes printing instructions obtained from a source other than the master. For example, the external instructions might come from the communications protocol used to transmit the master to the printer. The execution of the instructions body must leave on the stack one or more property vectors that will be combined using MERGEPROP to obtain the master's printing instructions. These are merged with external instructions, content instructions, default values, and printer overrides to produce the final instructions vector. The details of printing instructions are given in §3.3 and in Appendix E.

After the instructions body, if any, is executed, the top level block is executed. First, the block's preamble is executed with an initial frame containing *topFrameSize* zeros, and no arguments on the stack. The preamble returns no result, but the value of its frame after execution is used as an initial frame for each content node.

After the preamble is executed, each node of the block is executed in turn. If the node is a block, the execution scheme described here recurs. If the node is a body, it is executed with the initial frame supplied by the preamble of its block, and no arguments.

Bodies which are not instructions bodies, content instructions bodies, or preambles are *page image* bodies. A fresh printing surface is supplied for the image produced by each page image body.

Each page is produced by a distinct node in the skeleton. Hence an arbitrary number of pages cannot be printed by a loop; instead, each page must be represented separately in the master. This requirement is imposed to allow printing of selected pages and to allow the pages to be processed in an order that is convenient for the printer.

The skeleton structure allows the printer to reorder the actual printing of the individual page images for its convenience. Inside a body, anything can happen and no analysis is possible, in general, without executing the body in the proper state. In the skeleton, on the other hand, the relations among the parts are highly constrained, and analysis is easy. The skeleton is also useful for:

- simplifying the merging of masters (in many common cases);
- identifying parts of masters for printing, or for taking masters apart in ways not defined by Interpress;
- allowing definitions common to part of a master to be made local to that part.

The rules for executing nodes make it easy to merge masters, or to embed one master in another. A master *A* can be embedded as a node in another master *B*, simply by stripping off the *instructionsBody*. Its preamble will presumably make no use of the non-null initial frame it is given. More complex merging must be done by building new bodies. This can always be done by embedding existing bodies in new ones, and does not require going inside the existing bodies.

Note that an encoding can allow a node to have an external reference to a master so as to embed the contents of the master being referenced at that point (e.g., *sequenceInsertMaster*).

All the bodies in the skeleton are executed by DOSAVEALL, with a mark protecting the stack. Thus every body in the skeleton is executed with the imager variables in their initial state. This guarantees that executing a body cannot have side effects, and thus page image bodies can be executed in any order without affecting the output or each other. A consequence is that the preamble cannot set up imager variables for the page image bodies. Each body must do this for itself, perhaps by calling a composed operator constructed by the preamble.

No results may be returned by a page image body; its entire effect is captured in the image it produces.

The formal definition of the skeleton is given in the following Backus-Naur Form:

skeleton	::= instructionsBody block | block
instructionsBody	::= body
block	::= BEGIN preamble nodelist END
preamble	::= body
nodelist	::= node | node nodelist
node	::= content | CONTENTINSTRUCTIONS contentInstructionsBody content
content	::= block | body
contentInstructionsBody	::= body
body	::= *a body literal* as defined in §2.2.5

An example of a skeleton for a two-page master is {*instructionsBody*} BEGIN {*preambleBody*} {*pageImageBody*} CONTENTINSTRUCTIONS {*contentInstructionsBody*} {*pageImageBody*} END.

The meaning of a master (i.e., the output generated from it) is defined by the following pseudo-Pascal program which executes the skeleton. This program defines the *effect* of executing a master, not necessarily the implementation that a printer must use. In particular, Interpress is defined so that the master need not be executed *n* times in order to print *n* copies. The program intermixes Interpress operators with Pascal in an obvious way and treats Pascal local variables as elements of an Interpress frame, which it references with FGET and sets with FSET. It assumes a collection of global variables which may be accessed by the operators *PGET and *PSET.

It also uses several other special operators which do not exist in true Interpress. A description of these special operators follows:

<*j*:Cardinal> *PGET → <*value*: Any>
where the value of the j[th] global variable is placed on the stack.

<*x*: Any> <*j*: Cardinal> *PSET → < >
where the value of the j[th] global variable is set to x.

<*x*: Any> *ISBODY → <*i*: Cardinal>
where *i* is true (1) if *x* is a body, false (0) otherwise.

<*f*: Vector><*b*: Body> *MAKECOWITHFRAME → <*o*: Operator>
where *o* is a composed operator with body *b* and initial frame *f*.

<> *LASTFRAME → <*frame*: Vector>
where *frame* is the final value of the frame for the most recently executed composed operator.

<*m*: Vector> <*pageNumber*: Cardinal> <*duplex*: Cardinal> <*xImageShift*: Number> *SETMEDIUM → <>
where the effect is to start a new page. This operator is defined fully in §4.2.

<> *OBTAINEXTERNALINSTRUCTIONS → <*externalInstructions*: Vector>
where *externalInstructions* is a property vector representing printing instructions supplied to the printer from outside.

<*masterInstructions*: Vector> <*externalInstructions*: Vector> *ADDINSTRUCTIONDEFAULTS → <*finalInstructions*: Vector>
where *finalInstructions* is a property vector containing the printing instructions for the document. Generally, the effect is to provide default values for instructions which are not specified, to resolve conflicts between instruction values in *masterInstructions* and *externalInstructions*, and to impose printer overrides for those instructions for which the printer can support only specific values. The behavior of the operator is described in §3.3.1 and in the descriptions of the individual instructions in §3.3.3 and Appendix E.

*RUNSIZE and *RUNGET are defined in §3.3.2.

<*newContentInstructions*: Vector> <*oldInstructions*: Vector> <*externalInstructions*: Vector> *MERGECONTENTINSTRUCTIONS → <*newInstructions*: Vector>
where *newInstructions* is a property vector containing the printing instructions to be used for printing the associated content. Generally, the effect is to merge new content instructions with instructions from enclosing blocks and to revolve conflicts between content instruction values and instruction values found in the external instructions. The behavior of the operator is described for each content instruction in §3.3.4.

The execution of a master is defined by two procedures, *DoMaster* and *DoBlock*. These procedures share the global variables *outputOK*, *lastFrame*, *copyNumber*, *copyName*, and *pageNumber*. Operators that generate output (those with MASK in their names) check the value of *outputOK* and cause an error if it is zero. *DoBlock* saves in *lastFrame* the final value of a preamble body's frame (or the initial frame of a block node). These programs treat a block as a property vector with two elements: *preamble*, the preamble; and *nodes*, a Vector of nodes. A node is a property vector with two elements: *contentInstructionsBody*, which is null if no CONTENTINSTRUCTIONS token appears; and *blockOrBody*, the content of the node.

A master is executed by the *DoMaster* procedure, which takes the instructions body and the top-level block as arguments. It executes the *instructionsBody* with the *externalInstructions* vector of the environment as its argument. This leaves on the stack an instructions vector which is the master's estimate of the most appropriate instructions. There may be, however, external instructions which should take priority. This is accomplished by placing the external instructions on the stack once again, and executing the internal procedure *ADDINSTRUCTIONDEFAULTS. This operator is described in §3.3.1. The final instructions are recorded in the *instructions* variable. After all instructions processing, *DoMaster* calls *DoBlock* for each copy to execute the block with an initial frame with *topFrameSize* zero elements, and with the instructions.

global *variables.*
outputOK: Cardinal; *lastFrame*: Vector; *copyNumber*: Cardinal;
copyName: Any; *pageNumber*: Cardinal;

```
procedure DoMaster (instructionsBody: Body; topBlock: Block); begin
    instructions: Vector;
    iFrame: Vector [0 . . . topFrameSize-1] of Any;
    i, copyNumber: Cardinal;
    -- initialize the initial frame --
    for i := 0 to topFrameSize-1 do iFrame[i] := 0
    -- compute printing instructions --
    0 MAKEVEC instructions FSET -- null instructions in case mark recovery occurs --
    0 MARK -- protect against error while executing instructions body --
    0 outputOK *PSET
    *OBTAINEXTERNALINSTRUCTIONS -- get external instructions --
    iFrame FGET instructionsBody FGET
    *MAKECOWITHFRAME DOSAVEALL -- execute instructions body --
    while COUNT>1 do {MERGEPROP} -- merge all instructions for all elements on stack --
    instructions FSET
    UNMARK0
    instructions FGET
    *OBTAINEXTERNALINSTRUCTIONS -- get external instructions --
    *ADDINSTRUCTIONDEFAULTS -- install overrides and defaults --
    instructions FSET -- and save as final printing instructions --
    -- make the required number of copies --
    0 MARK -- protect against error anywhere in master, e.g., preamble --
    for copyNumber := 1 to instructions FGET copySelect GETP *RUNSIZE do
        { copyNumber FGET copyNumber *PSET -- set copyNumber --
        -- set copyName --
        instructions FGET copyName GETP copyNumber FGET *RUNGET copyName *PSET
        0 pageNumber *PSET
        if instructions FGET copySelect GETP copyNumber FGET *RUNGET then
            -- call DoBlock to execute the Block and print the copy --
            {DoBlock (topBlock FGET, iFrame FGET, instructions FGET)}
        }
    UNMARK0
end
```

A block is executed by the *DoBlock* procedure, which takes three arguments: the block, the initial frame, and the instructions vector. If the block argument is a body, *DoBlock* executes it with the given initial frame. If it is a block, *DoBlock* first executes the block's preamble using the initial frame argument. *DoBlock* calls itself recursively to execute the nodes of the block, this time using the initial frame determined by the preamble. For each new imaging surface, the *DoMediumSet* procedure is called to establish the medium values.

```
procedure DoBlock (block: Any; iFrame: Vector; instructions: Vector); begin
    i, numNodes: Cardinal; node: Any; tempInstruction: Vector;
    if block FGET *ISBODY then
        { 1 outputOK *PSET
        -- put body on proper page --
        pageNumber *PGET 1 ADD pageNumber *PSET
        if instructions FGET contentPlex GETPROP NOT
            { instructions FGET plex GETP } IF
            duplex EQ -- determine if duplex --
            instructions FGET pageOnSimplex GETPROP NOT -- see if pageOnSimplex
            instruction --
            { instructions FGET onSimplex GETP pageNumber *PGET *RUNGET } IF -- if not, just
            use onSimplex --
```

```
            OR -- the duplex and onSimplex tests --
            instructions FGET contentPageSelect GETPROP
            { copyNumber *PGET *RUNGET } IFELSE
            { instructions FGET pageSelect GETP copyNumber *PGET *RUNGET pageNumber
            *PGET *RUNGET } IF
            AND
            then
            { DoMediumSet(instructions FGET)
            iFrame FGET block FGET *MAKECOWITHFRAME DOSAVEALL }
        }
    else
        { -- case of a block --
        -- execute the preamble --
        0 MARK
        0 outputOK *PSET
        iFrame FGET block FGET preamble GETP *MAKECOWITHFRAME DOSAVEALL
        *LASTFRAME iFrame FSET -- set initial frame --
        UNMARK0
        -- execute the nodes, one at a time --
        -- calculate the number of nodes --
        block FGET nodes GETP SHAPE numNodes FSET POP
        for i := 0 to numNodes - 1 do
            { -- loop to execute each node --
            -- save ith node in "node"--
            block FGET nodes GETP i FGET GET node FSET
            0 MARK -- protect against error while executing the node --
            0 MAKEVEC tempInstructions FSET
            0 MARK -- separate mark recovery for content instructions --
            0 outputOK *PSET
            iFrame FGET
            node FGET contentInstructionsBody GETP
            *MAKECOWITHFRAME DOSAVEALL
            while COUNT > 1 do {MERGEPROP}
            tempInstructions FSET
            UNMARK0
            tempInstructions FGET
            *OBTAINEXTERNALINSTRUCTIONS *MERGECONTENTINSTRUCTIONS
            tempInstructions FSET
            DoBlock (node FGET blockOrBody GETP, iFrame FGET, tempInstructions FGET )
            UNMARK0
            }
        UNMARK0
    }
end
```

The internal operator *MERGECONTENTINSTRUCTIONS merges new content instruction values with the instructions vector of the block in which the content occurs. This operator is described in §3.3.4.

The procedure *DoMediumSet* prepares a new instance of the media.

```
procedure DoMediumSet (instructions: Vector);
    -- set medium for this page --
begin
    instructions FGET media GETP -- obtain media vector --
    instructions FGET pageMediaSelect GETPROP NOT -- and the pageMediaSelect --
    { instructions FGET mediaSelect GETP copyNumber *PGET *RUNGET
    pageNumber *PGET *RUNGET } IF -- if no pageMediaSelect, use mediaSelect --
    GET -- medium description --
    pageNumber *PGET -- page number --
    instructions FGET contentPlex GETPROP NOT
    { instructions FGET plex GETP } IF
     duplex EQ -- true if duplex --
    instructions FGET xImageShift GETP -- value of xImageShift --
    *SETMEDIUM -- prepare a new image surface --
end
```

3.1.1 Operator restrictions†

Some parts of a master do not allow certain operators to be executed. There are three classes of primitive operators:

BASE Any operator in the base language (defined in Chapter 2).

IMAGE Any imaging operator (defined in Chapter 4).

WEAKIMAGE Any **IMAGE** operator that does not have MASK in its name and is not defined in terms of operators that have MASK in their names; i.e., **WEAKIMAGE** operators are those that generate no output.

The instructions and content instructions bodies may execute only **BASE** operators excluding **IFCOPY**. The preamble body may execute only **BASE** and **WEAKIMAGE** operators. A page image body may execute any operator. Note that the restrictions apply only to the execution of operators; for instance, the preamble can make composed operators which contain arbitrary **IMAGE** operators, although it cannot execute them.

3.1.2 Pages†

The Interpress skeleton is a hierarchical structure which allows blocks to be nested within blocks. The model for mapping this structure onto printing surfaces traverses the skeleton from left to right in a depth-first fashion. Each page-image body maps onto a new printing surface.

The printer must arrange the printing sequence so that a stack of pages has the page with *pageNumber*=1 "on top," *pageNumber*=2 underneath page 1, etc. The value of *pageNumber* is defined in the program in §3.1.

Depending on the way the printer handles the media, the order in which images are printed may be the same as the order in the stack, the reverse, or some more complex function.

3.1.3 Combining and modifying masters†

There are many ways to combine masters into a single larger master. At one extreme are simple concatenations of two masters, so that the combination produces all the pages of the first, followed by all the pages of the second. Another simple operation is to convert a master *A* into another master *B* in which each page of *B* contains two pages of *A* rotated 90 degrees and placed side by side; this is sometimes called "two-up" printing. At the other extreme are combinations that depend on elaborate conventions, such as how illustrations contained in one master are merged onto pages contained in another master.

In many common cases, masters can be combined simply by combining the skeletons. For example, to make a master *C* which consists of all the pages of *A* followed by all the pages of *B*, simply construct a new top-level block.

BEGIN {} *topA topB* END

where *topA* and *topB* are the top-level blocks of *A* and *B*. Printing instructions for *C* must be derived from instructions associated with *A* and *B*; in particular, it is desirable to retain media-selection information.

3.2 Environments and names

Although it is possible for an Interpress master to be completely self-contained, most masters refer to objects that are outside of the master itself. These objects are contained in the *environment* furnished by the printer. They are obtained by the primitive operators FINDOPERATOR (§2.4.5), FINDDECOMPRESSOR (§4.6), FINDCOLOR (§4.7.1), FINDCOLOROPERATOR (§4.7.1), FINDCOLORMODELOPERATOR (§4.7.1), and FINDFONT (§4.9).

It is desirable for a master to obtain the objects used repeatedly by its page bodies in the preamble and save them in the initial frame, since an implementation is likely to handle FGET much more efficiently than the FIND-style operators.

3.2.1 Universal names

A master obtains an object from a printer's environment using a *universal name*, which is a Vector of Identifiers such that the first identifier in the name is a *universal identifier*. A universal identifier is an identifier which some organization has registered in the *Interpress Universal Registry*; the organization thereby obtains authority to determine the structure and meaning of all universal names having that first, universal identifier. Because the first identifier is registered, the organization can choose the remaining identifiers without danger of conflicting with names chosen by other organizations.

Although universal names can be structured in arbitrary ways subject to the constraint that the first identifier be a registered universal identifier, it is intended that a hierarchical naming system be used for objects in the environment, much like the hierarchical file directory system of many computer operating systems. This naming convention provides for hierarchical distribution of naming authority, and thus allows for unlimited growth of the name space without any name conflicts and without the need for any central authority to assign names.

Orderly invention of hierarchical names requires the notion of a *registry* , an administrative entity with authority over a particular point in the hierarchical name space, operated by some organization or person with an interest in

keeping order at that point in the space. Such points themselves of course have hierarchical names. The Interpress Universal Registry is responsible for assigning universal identifiers. The registry for point *n* is responsible for assigning names which begin with n so that conflicts do not arise. The registry can delegate part of its authority to a sub-registry, e.g., for the point *n/m*; in doing so, it assigns its own authority to invent names which begin *n/m*, since it cannot be sure that the sub-registry has not already chosen such a name.

By way of example, consider how Xerox might name fonts. The organization obtains the universal identifier *Xerox* from the universal registry. A registry for *Xerox* is established, responsible for assigning names that begin *Xerox/...*. The registry might choose to use character-set names at the next level, and invent the names *Ascii, Ebcdic, xc2-0-0,* etc. At the next level, type-family names might be used, e.g., *Xerox/xc2-0-0/Times*. Note that because the first identifier is registered, any names the Xerox registry invents are distinct from names assigned to other organizations, e.g., *Xerox/Ascii/ . . .* is distinct from *Mergenthaler/Ascii/ . . . .*

An organization wishing to name objects in the environment so that they can be referenced reliably from any Interpress master should obtain an identifier in the Interpress Universal Registry, and thus establish a name space within which to invent names for these objects. (See Appendix C.)

One of the identifiers registered in the Interpress Universal Registry is the identifier *standard*. This identifier is used to create universal names for objects which are commonly in the environment and for naming additional printing instructions and printing instruction values which will be widely used. Individuals or organizations wishing to have *standard* universal names assigned should contact the Interpress Registry. (See Appendix C.)

3.2.2 Environment names

The objects within the printer's environment may be organized according to their type in order to aid the FIND-style operators. The organization can be described by extending the object naming. There is a unique *environment name* for each object in the printer's environment that may be returned by a FIND-style operator. The environment name is obtained by prefixing the universal name with an identifier that describes its type. For example, the font with universal name *Xerox/Ascii/Times* has *fonts/Xerox/Ascii/Times* as its environment name. The prefixes are shown in Table 3.1.

Table 3.1 **Environment name prefixes**

Type of object	Operator that finds it	Prefix
operator	FINDOPERATOR	*ops*
decompression operator	FINDDECOMPRESSOR	*decompressionOps*
constant color	FINDCOLOR	*colors*
color operator	FINDCOLOROPERATOR	*colorOps*
color model operator	FINDCOLORMODELOPERATOR	*colorModelOps*
font	FINDFONT	*fonts*

3.2.3 Universal property vectors

Property vectors are often formed from a set of standard properties (e.g., for printing instructions). This is done in cases where it is important that the printer "understand" the properties in the vector. A limitation on possible property names is useful for this purpose. A *universal property vector* is a form of property vector with property names that are standard property names or universal names (§3.2.2) that allow growth of the name space. That is, a property name in a universal property vector must be either:

- an Identifier from a set defined by Interpress or other defining authority for this purpose, or
- a Vector of Identifiers, which specifies a universal name.

3.3 Printing instructions

In addition to the preamble bodies and page image bodies which define the page images which constitute a document, an Interpress master may also contain *printing instructions* which describe how the document should be printed. Printing instructions provide:

- specification of which pages are part of which copies,
- selection of which copies to print,
- finishing specifications,
- a description of supplemental pages such as the break page,
- information regarding resources required to print the master, and
- administrative information.

There are normally two sources for printing instructions: the execution of the master's instructionsBody and contentInstructions bodies, and external to the master by means such as the printing protocol. For example, if a master is stored and later printed on demand, some of the printing instructions, such as the number of copies to print, may be generated when the demand is made. Other instructions, such as the name of the document, are attached to the master when it is created. Interpress provides a flexible way of combining externally supplied instructions with those contained within the master.

Printing instructions are represented as a universal property vector in which property names denote particular printing instructions. The standard instructions are defined in this document, but printing instructions may also include printer-dependent or private instructions with universal names. With this generality, it is likely that not every instruction is understood by every printer. If a printer does not recognize a printing instruction, it simply ignores it.

As an example, the instructions vector [*docName*, <Report>, [*Xerox*, *offsetStacking*], *docOffset*] contains a standard instruction (*docName*) and an instruction that has been defined by Xerox, whose name is *Xerox/offsetStacking* and whose value is the identifier *docOffset*.

The Interpress Universal Registry contains the universal identifier *standard* which may be used to create universal names for additional printing instructions and printing instruction values which will be widely used. Individuals or

organizations wishing to have *standard* universal names assigned should contact the Interpress Registry. (See Appendix C.)

This section describes how printing instructions are computed and how a final set of instructions is resolved from instruction requests coming from the master, external sources, and the printer. It also describes the printing instructions which are directly involved in processing the Interpress skeleton, and which influence the appearance of the document. These are instructions recognized by all Interpress printers. The remaining instructions provide administrative and finishing information and are described in Appendix E.

It is not necessary to specify the value of every instruction because default values are provided by *ADDINSTRUCTIONDEFAULTS if needed. A value supplied by the master serves as a default to be used in the absence of a value from the external instructions. If a value is present in the external instructions, then it takes priority over a value in the instructions body. This means that one should not send "default" values in the external instructions.

3.3.1 Computing the printing instructions

When a master is executed, a property vector of *external instructions* is supplied by the printer to the master. Interpress does not define how the external instructions vector is obtained. The vector may contain instructions supplied as printer defaults, instructions obtained via the protocol used to trasmit the master to the printer, instructions supplied by the printer operator, or the vector may be empty.

A second set of printing instructions is obtained by executing the master's instructions body (if any). This body is executed with the external instructions vector on the stack. After the instructions body is executed, the contents of the stack are merged, using MERGEPROP, to obtain a single instructions vector.

After this instructions vector is obtained from the master, the *ADDINSTRUCTIONDEFAULTS operator is executed to compute the printing instructions which apply to the printing of the job as a whole and to establish the instructions vector used for the printing of the top level block of the master. (The effect of content instructions is described in §3.3.4). This step:

- provides default values for any needed instructions which are not specified by either the master's instructions vector or the external instructions vector,
- resolves values for instructions which occur in both the master's instructions vector and the external instructions vector, and
- imposes printer overrides for those instructions for which the printer can support only specific values.

With the exception of *pageSelect*, conflicts between instructions defined in this standard which occur in both the master's instructions vector and the external instructions vector are resolved by giving precedence to either the master or the external instructions. This precedence is specified with the instruction descriptions in §3.3.3 and Appendix E.

In the simplest case, instructionsBody simply pushes a single property vector on the stack, which is merged with the external instructions vector. For example, the external instructions vector specifies the number of copies, while the instructionsBody specifies the document name.

The instructionsBody can manipulate the external instructions vector in arbitrary ways. It may examine elements using GETPROP or GET. It may pop the external instructions vector off the stack and construct an entirely new set of instructions. Although instructions body is free to compute arbitrary instructions, *ADDINSTRUCTIONDEFAULTS has ultimate control, and may for example reinstate critical external instructions that instructions body deletes.

3.3.2 Run vectors

Some printing instruction values are vectors which may contain a great many elements. In order to keep these vectors compact, a run-length encoding scheme is used. The term *"Run of X"* refers to a vector that contains alternate values of type Cardinal and type X. The Cardinal specifies the number of times the value of X is repeated in the fully-decoded vector; the fully-decoded vector has a lower bound of 1. It is a master error if any of these vectors has an odd number of elements or has a non-zero lower bound. Except for *copySelect*, it is not an error if the fully decoded vector has more elements than are required to represent the information for a particular document or printing request, but a master error occurs if the vector has too few elements to represent the information.

For example, a Run of Identifiers vector might look like [2, *archive*, 3, *distribute*], which represents the full vector <*archive archive distribute distribute distribute* 1 5 MAKEVECLU>. The Run of Cardinals [1, 1, 6, 0, 1, 1] represents the full vector <1 0 0 0 0 0 0 1 1 8 MAKEVECLU>.

The precise treatment of run encoded vectors is described by two operators:

<*r*: Vector> *RUNSIZE → <*s*: Cardinal>
where *s* is the number of elements in the fully decoded form of the Run of X vector *r*. The effect of this operator is defined by the following program:
begin *l*,*n*,*j*,*s*: Cardinal; *c*: Any;
r FGET SHAPE *n* FSET *l* FSET -- *get lower bound and size of r* --
if *l*≠0 **or** *n* **mod** 2≠0 **then error**;
0 *s* FSET
for *j* := 0 **to** *n*–1 **by** 2 **do**
{ *r* FGET *j* GET *c* FSET -- *c* := *r[j]* --
if *c* FGET is not a Cardinal **then error**;
s FGET *c* FGET ADD *s* FSET -- *s* := *s*+*c* --
}
s FGET
end

<*r*: Vector> <*i*: Cardinal> *RUNGET → <*value*: Any>
where *value* is the i^{th} element of the fully decoded form of the Run of X vector *r*, 1≤*i*≤(*r* *RUNSIZE). The effect of this operator is defined by the following program:
begin *j*: Cardinal; *c*: Any;
if *i*=0 **or** *i*≥ (*r* *RUNSIZE) **then error**;
for *j* := 0 **by** 2 **do**
{ *r* FGET *j* FGET GET *c* FSET -- *c* : = *r[j]* --
if *c* FGET is not a Cardinal **then error**;
if *c* FGET ≥ *i* **then goto** *done*;
i FGET *c* FGET SUB *i FSET*; -- *i* : = *i* - *c* --
} **error**;
done: *r* FGET *j* FGET 1 ADD -- *value* := *r[j*+*1]* --
end

3.3.3 Skeleton instructions

The following printing instructions are directly involved in the processing of the Interpress skeleton and influence document appearance. They are supplied either in the external instructions or in the master's instructionsBody. Each instruction description gives the instruction's name (i.e., the property name to use in an instructions vector), followed by its type (i.e., the type which the value corresponding to the instruction must have). If the type is

not correct, a master error results. In the text describing each instruction, the symbol *value* denotes the value associated with the instruction property.

With the exception of *pageSelect*, if an instruction's value is specified in both the master's instructions body and the external instructions *ADDINSTRUCTIONDEFAULTS establishes the value which is used by precedence. In the following definitions, the parenthesized note after the definition of the value's type specifies whether the value computed in the master (Master) or the value supplied by external instructions (External) is given precedence. The computation for *pageSelect* is given with the description of that instruction.

insertFileMapping: Vector of Mapping. (External)

This instruction gives mappings from names given in the master to names that should be used instead. Each Mapping is a Vector [*master, actual*], where both *master* and *actual* are Vectors of Cardinals. This requires any *sequenceInsertFile* token or *sequenceInsertMaster* token whose data bytes (as 8 bit binary numbers) are equal to the values of the elements of *master* to be replaced by a sequence of bytes which are the values of the elements of *actual* (mod 256).

media: Vector of Property Vector. (External)

This instruction describes each medium used to print the master. Each element of the *media* vector is a *MediumDescription*, which is a universal Property Vector. The *media* vector is indexed by a *MediumIndex* to select a *MediumDescription*. The property names for *MediumDescription* are:

mediumName: Identifier or Vector of Identifier. The *mediumName* property normally identifies a standard medium or a form which can be completely specified by a single name. In this case, the other properties of the medium are inferred from it. The identifier value for *mediumName* is *defaultMedium*, which designates a default medium chosen by the printer.

mediumXSize, *mediumYSize*: Number. The size of the medium in meters, as defined in §4.3.1.

mediumMessage: MessageString. A message which may describe the medium, to be delivered to the operator prior to and as close as possible to master imaging. MessageStrings are described in Appendix E.2.

Vector of Identifier. Additional properties may be designated by universal names.

Default: [[*mediumName*, *defaultMedium*]].

plex: Identifier. (External)

This instruction specifies how the page images are placed on instances of the media (e.g., sheets of paper). The identifiers that may appear as values for this instruction are:

simplex: One page image is placed on each instance of the medium.

duplex: Two page images are placed on each instance of the medium using printing surfaces on both sides of the medium. The orientation of the printing surface used for the second page image is obtained by rotating the medium about the *y* axis of the Interpress coordinate system.

Default: Either *simplex* or *duplex*, as chosen by the printer.

xImageShift: Number. (External)
The value specifies the distance (in meters) that each page's image is to be shifted in the *x* direction. On odd (*pageNumber* **mod**2 = 1) pages, the shift is to the right if value > 0 or to the left if value < 0. On even (*pageNumber* **mod**2 = 0) pages, the shift is to the left if value > 0 or to the right if value < 0. This effect is stated precisely in §4.2. Default: 0.

The following printing instructions specify the configuration of pages and copies to print. Each page number *p* of each copy number *c* may receive different treatment. The page (*c,p*):

- is printed only if *copySelect*[*c*] ≠ 0 **and** *pageSelect*[*c,p*] ≠ 0 **and** (printing duplex **or** *onSimplex*[*p*] ≠ 0);
- uses the medium identified by the *MediumIndex* equal to *mediaSelect*[*c,p*];
- has a copy name equal to *copyName*[*c*]; the copy name is used in conjunction with IFCOPY, §2.4.7.

The first two decisions may be altered by the use of content instructions (§3.3.4).

The arrays *copySelect*, *pageSelect*, *onSimplex*, *mediaSelect*, and *copyName* are provided in their respective printing instructions using the run encoding scheme. They are used to define the treatment of all pages and copies:

copySelect: Run of Cardinal. (External)
Default: [1, 1], which selects a single copy.

copyName: Run of Identifier. (External)
Default: [10^7, *null*], which provides the default name of *null* for every copy.

mediaSelect: Run of (Run of Cardinal). (External)
The outside vector is indexed by copy number, the embedded vector by page number. The Cardinal value is a *MediumIndex* which selects a *MediumDescription* from the *media* vector.

Default: [10^7, [10^7, 1]], which selects the medium with *index = 1* in the value of the *media* instruction for all pages and all copies.

onSimplex: Run of Cardinal. (External)
Default: [10^7, 1], which prints all pages when printing *simplex*.

pageSelect: Run of (Run of Cardinal).
The outside vector is indexed by copy number, the embedded vector by page number.

Default: [10^7, [10^7, 1]], which selects all pages for all copies.

If *pageSelect* = *masterPageSelect* in the master's instructions vector and *pageSelect* = *externalPageSelect* in the external instructions vector, the resultant value is

page Select(c.p) = *masterPageSelect(c.p)* AND *external PageSelect(c.p)*

The *copySelect* instruction requests both the number of copies and the copy numbers for those copies. For example, the instruction [*copySelect*, [5, 1]] prints copy 1, copy 2, copy 3, copy 4, and copy 5. Note that copy 1 and copy 2 may differ because of the way *pageSelect* and *mediaSelect* instructions are given or because of the action of IFCOPY. As another example, the instruction [*copySelect*, [2, 0, 1, 1]] causes only copy 3 to be printed.

The normal convention for *copyName*, *pageSelect*, and *mediaSelect* is to use a run representation that provides instructions for a great many copies (e.g., 10^7), far more than will ever be selected for printing by *copySelect*. For example, the instruction [*pageSelect*, [10^7, [5, 1]]] selects pages 1 through 5 for printing on all (conceivable) copies.

If the instructionsBody is constructed before the total number of pages in the document is known, the run representation for pages may also be chosen to exceed the actual number of page image bodies in the master. Thus [*onSimplex*, [10^7, 1]] prints all pages when printing *simplex*, even though the document will not have 10^7 pages.

3.3.4 Content instructions

Certain printing instructions may be specified within a contentInstructions body rather than in the master's instructionsBody. These instructions augment or override the instructions found in the master's instructionsBody and do not have default values; if absent, the default behavior is described by the master's instructionsBody. Since contents may be nested within a master, a mechanism is needed to determine the net effect of contentInstructions for nested contents. This is done by *MERGECONTENTINSTRUCTIONS. The effect of the *MERGECONTENTINSTRUCTIONS on the instructions vector which is used for printing the content of the contentInstructions body's node is to:

- concatenate file names mappings as specified in the *contentInsertFileMapping* instruction,
- impose the instruction values specified by the content instructions body on the current instructions vector,
- to reimpose values specified in the external instructions which have precedence over the content instruction as specified in the content instructions description, and
- to reimpose printer overrides for those instructions for which the printer can support only specific values.

The resulting instructions vector is used for printing the content's page image body if the content consists of a single body or as the argument to *DoBlock* if the content is a block. The execution of the contentInstructions body places on the stack a property vector that may contain the following content instructions:

contentInsertFileMapping: Vector of Mapping.

A Mapping is a two element vector of the form [*master, actual*] where both *master* and *actual* are Vectors of Cardinals representing strings. They provide a mapping between a "master" name for the data bytes in a *sequenceInsertFile* or *sequenceInsertMaster*, and an "actual" name which is used outside of the content, as described previously for *insertFileMapping*.

A file name contained in a *sequenceInsertFile* or *sequenceInsertMaster* token undergoes consecutive mappings via all containing *contentInsertFileMapping* instructions, and the *insertFileMapping* instruction, in order to determine the final file name. The *MERGECONTENTINSTRUCTIONS procedure appends the value of this instruction to a vector of values of *contentInsertFileMapping* for containing contents. If an *insertFileMapping* instruction is present in the external instructions, it has no effect over the *contentInsertFileMapping* values, and only takes precedence over an *insertFileMapping* instruction in the master's instructionsBody.

contentPageSelect: Run of Cardinal. (External)

This instruction augments the *pageSelect* specification from the master's instructionsBody. The run vector is indexed by copy number and indicates whether the pages within the scope of the content should be printed for that copy. A non-zero value

means that the pages should be printed. The *contentPageSelect* can reselect pages which have been deselected by the *pageSelect* instruction or *contentPageSelect* instructions from enclosing contents.

If a *pageSelect* instruction is specified in the external instructions and a *contentPageSelect* instruction is present, then the page selection criteria for the content is:

pageSelect(c.p) = contentPageSelect(c) AND *external PageSelect(c.p)*

On *MERGECONTENTINSTRUCTIONS, the new value for this instruction is the value supplied by the *contentInstructionsBody*. The new value applies for the scope of the content.

contentPlex: Identifier. (External)

This instruction tells the printer whether the page image bodies within the scope of the content are printed *simplex* or *duplex*. The possible values are *simplex* and *duplex*. The value takes precedence over the master's *plex* instruction for the pages within the scope of the content.

If a *plex* value is supplied by the external instructions, it takes precedence over both the *plex* and the *contentPlex* values. In *MERGECONTENTINSTRUCTIONS, the new value of this instruction replaces any previous value for the scope of the content.

The following instructions are intended to apply to single page image bodies and normally precede contents which are bodies.

pageMediaSelect: Cardinal. (Master)

The value is a *MediumIndex*, which selects the *MediumDescription* from the *media* vector for use in printing the pages in the content. This value takes precedence over the value specified by *mediaSelect(c,p)* for the scope of the content, whether or not *mediaSelect* is specified in the external instructions. In *MERGECONTENTINSTRUCTIONS, the new value of this instruction replaces any previous value for the scope of the content.

pageOnSimplex: Cardinal. (Master)

The value is used as the *onSimplex* value for this content. This value takes precedence over the value specified by *onSimplex(p)* for the pages in the content whether or not *onSimplex* is specified in the external instructions. In *MERGECONTENTINSTRUCTIONS, the new value of this instruction replaces any previous value for the scope of the content.

The details of the processing of content instructions are shown in the program in §3.1.

Further content instructions are presented in Appendix E.4.

Note that content instructions may be used in conjunction with IFCOPY to obtain different effects on different copies.

As an example of content instructions, consider a master that is to print invoices using two pre-printed forms, one for the first page of every invoice and a second for continuation pages, if any. The creator cannot anticipate, when the instructions body is generated, precisely which pages in the master require which forms. Instead, the creator uses {[*pageMediaSelect*, 1]} or {[*pageMediaSelect*, 2]} as a content instructions body to select an appropriate medium for each page. Of course, the instructions body must contain an appropriate *media* entry, such as [*media*, [[*name*, [*Xerox, invoice-first*]], [*name*, [*Xerox, invoice-continuation*]]]].

Interpress operators which create images are called *imaging operators*; the operators are typically invoked when a page image body (§3.1) is executed. These operators are implemented by an *imager* program, which is called to produce the desired images. The discussion begins with several sections that outline general concepts of the operators, such as the *imaging model* and *coordinate systems*. These sections are followed by complete descriptions of the operators.

4.1 Imaging model

Interpress synthesizes a complex image by repeatedly laying down simple *primitive images*. For example, a single Interpress operator might place an image of a specific character at a specific position on the page. A subsequent operator might place another character somewhere else, and so on until a complex image is built up. A painter performs a very similar set of operations to create a complex image: he selects a brush, dips it in paint, and lays down a stroke of color. Complex images are created by a series of these simple actions.

The Interpress imaging model, illustrated in Figure 4.1, involves three objects:

- *The page image.* The two-dimensional page image accumulates the primitive images being laid down. It plays the role of the painter's canvas.

- *The mask.* The mask specifies the shape and position of a primitive image to be added to the page image; it determines exactly where the page image will be modified. The illustration shows a character 'b' whose shape and position are described by the mask. In effect, the mask specifies an opening through which ink can be pressed onto the page image. The mask thus plays the role of the painter's brush stroke.

- *The color.* The color specifies the ink to be pushed through the mask onto the page image in order to add the primitive image to the page image; it may take on many colors, various shades of gray (including white and solid black), and transparent. To continue the painting analogy, the color specifies the color of paint in which to dip the brush.

Interpress makes complicated images, then, by specifying a sequence of (mask, color) pairs to be applied to the page image. Invocations of mask operators actually cause the page image to be altered. The color is held in a state variable in the imager; it applies to all masks until changed.

At the beginning of each page, a new page image is initialized to "paper-white," or the natural color of the material on which the image is being formed. The imaging operators control the ink deposited on the material. In other imaging applications, the initial state of the page image is chosen to achieve as nearly as possible the same effect. If images are being made on a television display, the initial state of the entire display is white. If positive images are being made on film, the initial state of the film is transparent, which corresponds to

white if the film is projected with a white light. Although the discussion of Interpress frequently refers to "pages" and "ink" for convenience, Interpress imagers can be used to create images on any two-dimensional medium.

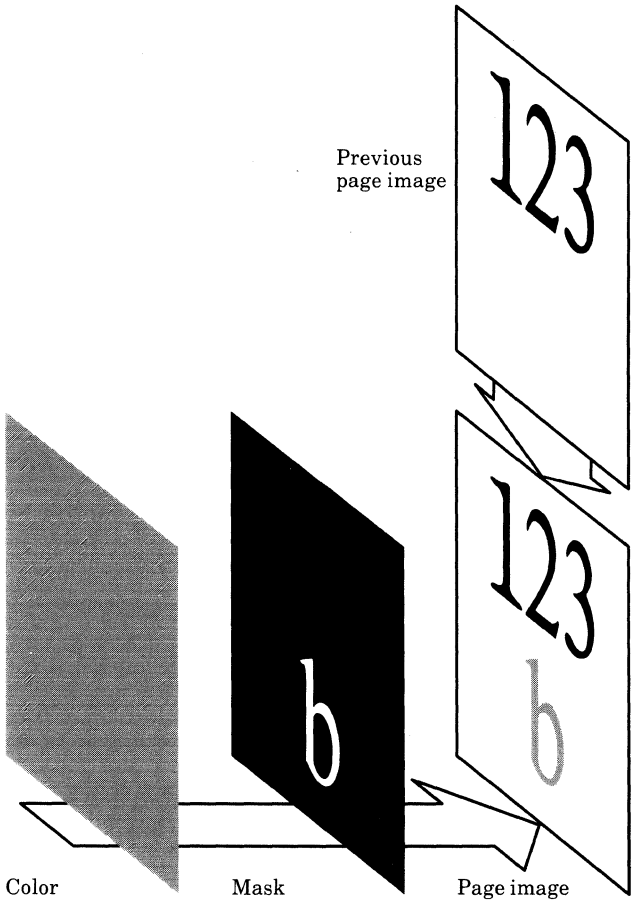


Figure 4.1 **The imaging model; application of a mask operator**

4.1.1 Priority

The repeated generation of primitive images raises an important question: when two primitive images overlap on the page image, which one is visible? The answer is the intuitive answer used by the painter: laying down an object obscures any overlapping parts of objects that have been previously placed on the page image, unless the color is transparent. This phenomenon is called *priority*: objects laid down later have priority over objects laid down earlier. The order of specification of the objects (i.e., the order of execution of the mask operators) determines the *priority order*.

There are times when priority order does not matter. For example, if all objects are the same color, reordering their priority does not change the image: objects that overlap simply appear to merge. Priority is also unimportant when objects do not overlap, regardless of their colors. In fact, priority order is important only when objects of different colors overlap.

If a master wishes the imaging operators to preserve the priority order of objects, it must set the imager variable *priorityImportant* to a non-zero value (§4.2). A change to the page image induced by a mask operator is said to be *ordered* if the mask operator is executed when

priorityImportant is not 0, and *unordered* if *priorityImportant* is 0. The rule is that the priority order of all ordered page-image changes must be preserved. The imager is allowed to alter priority order among unordered changes, or between ordered and unordered changes. Because preserving priority order may require more computation than allowing arbitrary reordering of objects, creators should leave *priorityImportant* 0 if possible.

4.2 Imager state

The state of the imager is contained in two places: (1) the page image itself, which is inaccessible to operators specified in the Interpress master, and (2) two sets of variables: the *persistent* imager variables and the *non-persistent* imager variables. Operators are provided for reading or writing these variables; in examples they are called with the name of the variable, but the master must call them with the variable's Cardinal index.

<*j*: Cardinal> IGET→ <*x*: Any>
where *x* is the value of the variable with index *j* in Table 4.1. A master error occurs unless *j* appears in the Index column of the table.

<*x*: Any> <*j*: Cardinal> ISET→ < >
where the value of the variable with index *j* in Table 4.1 is replaced by *x*. A master error occurs unless *j* appears in the Index column of the table. The type of *x* must match the type of the imager variable indexed by *j*, as given in Table 4.1.

Each variable is assigned an initial value (see Table 4.1) when the interpretation of an Interpress master begins. The imaging operators make frequent use of the variables; some operators change their values.

The *SETMEDIUM operator, called at the beginning of a page, alters the imager state to establish the imaging medium. Specifically, its actions are:

<*m*: Vector> <*pageNumber*: Cardinal> <*duplex*: Cardinal> <*xImageShift*: Number> *SETMEDIUM→ < >
where an instance of the available physical medium which most closely approximates the medium specified by the MediumDescription *m* (§3.3.3) is selected and certain imager variables are set:

- Set *mediumXSize* and *mediumYSize* appropriately for the physical medium selected. Set *fieldXMin*, *fieldXMax*, *fieldYMin*, and *fieldYMax* as well (§4.3.1).

- Set $T := <S\ T_{ID}$ CONCAT>, where T_{ID} is the ICS-to-DCS transformation for this page (see §4.3 and §4.4, especially §4.4.5). The transformation *S* is a translation transformation determined by the *xImageShift* printing instruction. *S* := **if** *duplex* **and** *pageNumber* **mod** 2=0 **then** <-*xImageShift* 0 TRANSLATE> **else** <*xImageShift* 0 TRANSLATE>.

If *xImageShift* is used, the imager variables *mediumXSize*, *fieldXMin*, and *fieldXMax* specify a smaller useful *x* dimension than normal. That is, the *x* size is reduced by the absolute value of the image shift. These details are omitted in the description above.

Table 4.1 **Imager variables**

Name	Index	Type	Section	Initial value
Persistent:				
DCScpx, DCScpy	0, 1	Number	4.5	0, 0
correctMX,correctMY	2, 3	Number	4.10	0, 0
Non-persistent:				
T	4	Transformation	4.4	<1 SCALE> (identity)
priorityImportant	5	Cardinal	4.1.1	0
mediumXSize, mediumYSize	6, 7	Number	4.3	0, 0
fieldXMin, fieldYMin	8, 9	Number	4.3	0, 0
fieldXMax, fieldYMax	10, 11	Number	4.3	0, 0
font	12	Font	4.9	<[ ] MAKEFONT>
color	13	Color	4.7	<1 MAKEGRAY> (black)
noImage	14	Cardinal	4.8	0
strokeWidth	15	Number	4.8	0
strokeEnd	16	Cardinal	4.8	0 (*square*)
strokeJoint	23	Cardinal	4.8	0 (*miter*)
underlineStart	17	Number	4.8	0
amplifySpace	18	Number	4.9	1
correctPass	19	Cardinal	4.10	0
correctShrink	20	Number	4.10	$\frac{1}{2}$
correctTX, correctTY	21, 22	Number	4.10	0, 0
clipper	24	Clipper	4.8	full field

The variables differ in their treatment by the DOSAVE operator. None are restored at the completion of the DO operator; the non-persistent variables are restored by DOSAVE; all are restored by DOSAVEALL.

4.3 Coordinate systems

Locations on the page image are denoted by (x, y) coordinates in a corresponding *coordinate system*. Ultimately, all locations must be converted into the *device coordinate system* (DCS), the coordinate system used by the imaging device producing the page image. However, if the master were to specify coordinates directly in the device coordinate system, the master would not be device-independent, and could not be used for other devices with different coordinate systems. A device-independent *Interpress coordinate system* (ICS) is introduced in order to establish positions and sizes of objects on a page image independent of the resolution of the printing device. The imager operators convert from the ICS to the DCS as an image is formed.

4.3.1 Medium size and orientation

The medium used to print a page is chosen by printing instructions (§3.3). The orientation of the ICS on the medium is such that the y axis points up and the x axis to the right in the normal viewing orientation as defined below; see Figure 4.2. The physical size of the medium (in meters) is made available in the imager variables *mediumXSize* and *mediumYSize*; the choice of normal viewing orientation defines which is which. The *SETMEDIUM operator (which cannot be called from the master) sets the *medium...* and *field...*

variables before a page body is executed. The master can read these six variables with IGET; any attempt to set them with ISET causes a master error.

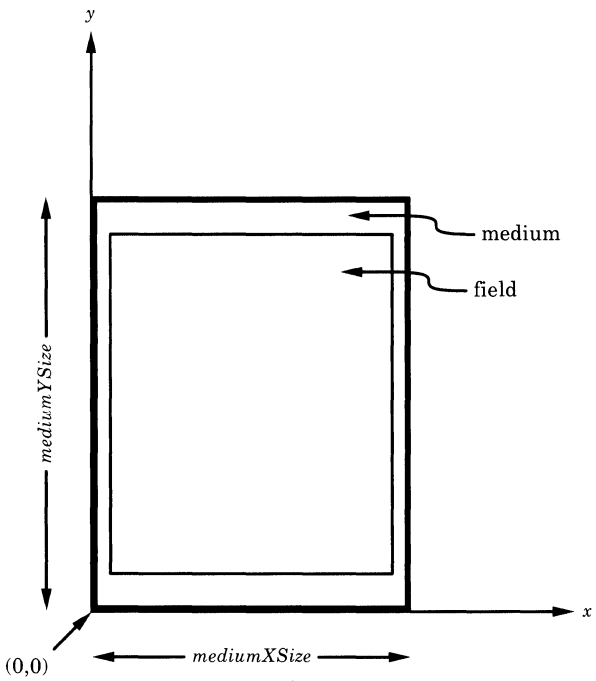


Figure 4.2 **Physical medium**

The normal viewing orientation is a property of the *medium*, not of the *image*. If the creator knows the orientation, it can orient the image on the medium as it likes by supplying a suitable transformation. To enhance device-independence, it is desirable for printers to choose the normal viewing orientation consistently. Often the choice is somewhat arbitrary (e.g., cut sheet paper which is not square can have either portrait or landscape orientation). In case of doubt it is recommended that the choice be made so that *mediumYSize* is greater than *mediumXSize* (portrait orientation). Creators can adapt to this convention by supplying a 90° rotation for images which are normally viewed with the long axis horizontal.

A conventional sheet of $8\frac{1}{2}\times 11$ inch paper could have either $mediumXSize = 0.0254\times 8.5 = 0.2159$ and $mediumYSize = 0.0254\times 11 = 0.2794$, or $mediumXSize = 0.2794$ and $mediumYSize = 0.2159$. In addition, for each of these choices there are two possible orientations of the axes on the sheet as it emerges from the printer. If the medium consists of separate sheets of paper, a sheet can always be rotated by the reader to assume the desired orientation. If the sheets are bound or stapled, however, or if the image is being displayed on an immovable display surface, rotation is impractical. For these reasons, a consistent choice of normal viewing orientation is quite important. However, it cannot be enforced by the Interpress standard because of the wide variation in the physical characteristics of printers and media. The rule given above yields $mediumXSize = 0.2159$ and $mediumYSize = 0.2794$. The choice between the two orientations which remain may be arbitrary, or it may be strongly suggested by the finishing method or other physical properties of the printer.

The two modes of placing page images onto two-sided media are *simplex* printing and *duplex* printing. In simplex printing one page image is placed on each instance of the medium (e.g., on each piece of paper). The side on which the image is placed is referred to as the *front* side of the page of the finished document. The (unbound) printed document is presented with the pages in the order presented in the Interpress master when viewed from the front side. Medium changes have no effect on on image placement or presentation order. In some cases, as for example with a pre-printed form, the medium itself may be considered to have a front side and a back side. In this case the side on which the image is placed may be either the front side or the back side of the unprinted medium as specified by printing instructions. Whichever side is used, the side on which the Interpress page image is placed is considered the front side for document assembly.

In duplex printing, the placement of page images on the media depends on the the medium selection as well as the sequence of page images. To determine the image placement, the page images in the document are first separated into *runs* based on the *mediaSelect* vector. Each run of page images is then placed on the medium as follows: the first page image of the run is placed on that side of the medium designated by the printing instructions as for simplex printing; the second page image is placed on the back of the same instance of the medium; the third page image is placed on the front of the second instance of the medium; and so on. If there are an odd number of page images in any run, no image is placed on the back of the last instance of the medium on which the page images in that run are being placed and the placement of images in the next run (if any) proceeds on a new instance of the appropriate medium. The (unbound) printed document is presented with the first, third, fifth, etc., page images of each run to the front of the printed document.

Each medium has an associated rectangular *field*, a portion of the medium in which imagery may lie. Ideally, the field would contain the entire medium, but some imaging hardware cannot place imagery along borders of the medium. The field in which imagery may lie is described by four numbers: $fieldXMin \leq x \leq fieldXMax$, $fieldYMin \leq y \leq fieldYMax$. These values are available as imager variables, set by *SETMEDIUM. Conventions, printing hardware, and imagers should strive to make the border areas as small as possible. For example, the field of an image on a television screen would fill the entire medium (i.e., the medium and field sizes would be identical).

4.3.2 Interpress coordinate system (ICS)

The ICS is a standard way to specify positions on the image, using the coordinate system origin and directions established for an image. The two coordinate axes of the ICS are named x and y. The rectangular image includes the origin and lies in the first quadrant. The units of measurement in the ICS are meters; coordinates in the ICS are represented by Numbers. The coordinate system is chosen so that the y axis points "up" in the normal viewing orientation. Coordinates in the ICS that are transformed to device coordinates must obey the restrictions specified in §5.2.

4.3.3 Master coordinate systems

Using transformation operators described in §4.4, the master may establish any number of coordinate systems in which it can express locations of objects. To establish such a coordinate system, the master specifies a transformation that converts from the master coordinate system into the ICS. The term *master coordinates* refers to coordinates that appear in the master, which will be transformed into the ICS by a transformation also specified in the master.

Master coordinate systems may be chosen so that all coordinates in the master can be conveniently represented as integers. Then a transformation is specified to convert to Interpress coordinates.

For example, the creator might choose to represent all coordinates in units of 1/10 printer's point, or 1/720 inch. For an $8\frac{1}{2} \times 11$ inch page, coordinates would lie in the range $0 \leq x \leq 6120$, $0 \leq y \leq 7920$. The transformation from master to ICS would scale by 0.0254/720.

4.3.4 Coordinate precision†

The precision with which an image can be created depends on the precision of the arithmetic used in the master and in the printer to describe shapes on the image. The precision also depends on the resolution of the imaging hardware. This section describes how Interpress controls the imaging precision. Informally, the precision must be at least as good as the precision of the imaging device, but is not required to be much better.

A master specifies an image by executing a sequence of primitive mask operators. Each of these has one or more coordinates as arguments; a transformation is applied to these coordinates. The mask operator arguments, once transformed, specify the shape and location of the mask on the image. The *ideal* image is the result of applying ideal mask operators, as defined in this document, to ideal coordinate arguments, computed by ideal arithmetic from the literal numbers included in the master. An image is a *good* image for a particular device if it is as close to the ideal image as any image that can be produced by executing the same sequence of mask primitives, subject to limitations imposed by the precision of mask operators and of the device resolution.

In order to describe the precision of the imaging computations, a grid is imposed on the Interpress coordinate system. It is a rectangular lattice of points, aligned with the x and y ICS axes, such that the ICS origin is at a grid point. The printer chooses values for g_x, the grid spacing along the x axis, and g_y, the (possibly different) grid spacing along the y axis; both are measured in meters. Grid points are thus located at all points (ng_x, mg_y), where n and m are Cardinals. A *spot* is the rectangular region bounded by adjacent grid lines; it has width g_x and height g_y.

The grid must have just enough grid points to match the spatial resolution of the imaging device. The idea is that a good image will be produced if each coordinate is rounded to the nearest grid point to determine where a mask should lie on the image.

If a raster-scanned device can present only bi-level images (e.g., black and white only), the grid spacing will be identical to the pixel spacing on the device. However, a device capable of showing several intensities at each point has an effective spatial resolution that is higher than the device's pixel resolution if images are properly filtered and sampled. In this case, the grid spacing may be smaller than the device's pixel spacing.

Interpress places no precise requirements on how masks are scan-converted, i.e., how the shape of a mask is converted into signals to control the pixels on the imaging device. The role of the grid in Interpress is to allow images to be aligned with the grid to be sure they have the same spatial configuration in all cases (see TRANS).

Interpress guarantees that coordinate arithmetic errors will never exceed a fraction of a spot dimension, provided the master obeys certain rules (§5.6).

4.3.5 Device coordinate system (DCS)†

The device coordinate system (DCS) is a transformation of the Interpress coordinate system chosen for the convenience of the printer. A transformation T_{ID} converts coordinates measured in the ICS to coordinates measured in the DCS. Device coordinates must have sufficient precision to represent grid points.

The ICS-to-DCS transformation T_{ID} is restricted to be of the form

$$x_{DCS} = \alpha(x_{ICS}/g_x + \gamma)$$
$$y_{DCS} = \beta(y_{ICS}/g_y + \delta)$$

subject to the following constraints:

- $|\gamma|$, $|\delta|$ are Cardinals
- There is some ICS point (x,y), $0 \le x \le mediumXSize$, $0 \le y \le mediumYSize$, that is mapped to the DCS point (0, 0).

The first constraint requires that (0, 0) in the DCS be a grid point. The second constraint ensures that no more precise arithmetic is required to represent the DCS than to represent the ICS (see §5.6).

It is often convenient if the device coordinate system is the same as the pixel addressing conventions for the printing device, i.e., $|\alpha|=|\beta|=1$. Thus grid points will have integer coordinates; this choice simplifies rounding to the nearest grid point when scan-converting masks.

All of the properties of a pixel addressing system for a device are captured in the Interpress-to-device transformation that converts ICS coordinates into DCS coordinates. Although the device coordinate system will have its axes aligned with those of the ICS, axis directions may vary. For example, a device coordinate system for a raster-scanned display might choose (0, 0) to be in the upper left-hand corner, with y increasing downward and x increasing toward the right in order to correspond to pixel-addressing hardware in the display. The ICS-to-DCS transformation can perform these conversions as well as the scaling of units.

A single special operator is associated with device coordinates:

<x: Number> <y: Number> *DROUND → <X: Number> <Y: Number>
where (X, Y) are the device coordinates of the grid point best representing the point with device coordinates (x, y).

4.4 Transformations

Linear transformations are used to map coordinates from one coordinate system to coordinates in another system. A transformation is used to map from the Interpress coordinate system to the device coordinate system, and one may also be used to map from a master coordinate system to the Interpress coordinate system. Transformations may be used freely in the master to establish master coordinate systems that are convenient for representing parts of the image.

A coordinate specified in the master may need to be subjected to several transformations in order to map it all the way into the device coordinate system. Fortunately, the effect of several transformations applied in sequence can be expressed as a single, combined, transformation. The Interpress imaging operators map every coordinate they are presented using the *current transformation* T, an imager variable. This transformation expresses the combination of all transformations that must be applied, including the ICS-to-DCS transformation. Operators are provided for changing the value of T.

A coordinate transformation M is represented as a 3×3 matrix M, interpreted as follows:

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

We speak of this transformation mapping coordinates in the *from* coordinate system to those in the *to* coordinate system. The matrix M has the form:

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

The last column of a transformation is *always* 0, 0, 1, and need not be explicitly stored. Moreover, in the computation of x_{to} and y_{to}, only the following computations are required:

$$x_{to} = ax_{from} + by_{from} + c$$
$$y_{to} = dx_{from} + ey_{from} + f$$

A transformation of this sort can represent scaling, rotation, translation, or combinations of these primitive transformations.

Often two transformations are *concatenated* to form a single composed transformation that achieves the same effect as applying the two in sequence. For example, suppose that the transformation T represents the transformation from the Interpress coordinate system to the device coordinate system:

$$[x_{DCS}, y_{DCS}, 1] = [x_{ICS}, y_{ICS}, 1]\, T$$

Now suppose coordinates are specified in a more convenient master coordinate system (c) and transformed by a transformation C from this system to the Interpress system:

$$[x_{ICS}, y_{ICS}, 1] = [x_C, y_C, 1]\, C$$

By substituting the second equation into the first, the two steps become:

$$[x_{DCS}, y_{DCS}, 1] = [x_C, y_C, 1]\, (CT)$$

where C and T have been multiplied together to form a single matrix. By concatenating pairs of transformations, a sequence of transformations can be represented as a single matrix. In this way an arbitrary coordinate system may be mapped directly to the device coordinate system.

Some printers may restrict the transformations which can be used with character operators and pixel arrays, as described in §5.4.2.

4.4.1 Instances of symbols

The operators for modifying the current transformation are designed to help make *instances* of *symbols*. For example, the graphical shape of a character is defined by a composed operator: this composed operator represents the character *symbol*. To cause a particular occurrence, or *instance*, of the character to appear on the page, the coordinates contained in the symbol definition must be transformed into the ICS: this transformation governs the size, orientation, and location of the instance. Then the coordinates in the ICS must be transformed by the Interpress-to-device transformation as well. Both steps are accomplished by replacing the current transformation matrix T, which initially contains the Interpress-to-device transformation, with a concatenated transformation CT, where C expresses the symbol-to-Interpress transformation; the new T achieves the combined effect of both transformations. The imaging operators in the symbol definition are now interpreted; all coordinate arguments are transformed by the new T. When the execution of the symbol is over, the value of T must be restored to the value in effect before the call. The primitive operator SHOW (§4.9) performs instancing of this sort; the saving and restoring of T is performed by DOSAVESIMPLEBODY.

4.4.2 Notation

Coordinate transformations are used extensively in the remainder of this section. The following notation is used for the two common kinds of transformation:

"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 $[X, Y, 0] = [x, y, 0]\ m$.

4.4.3 Transformation operators

<*a*: Number> <*b*: Number> <*c*: Number> <*d*: Number> <*e*: Number> <*f*: Number> MAKET → <*m*: Transformation>

where the transformation *m* is defined by the matrix:

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

<*x*: Number> <*y*: Number> TRANSLATE → <*m*: Transformation>

where the effect is 1 0 *x* 0 1 *y* MAKET. <*x y* TRANSLATE> will map the origin to the point (x, y). The transformation *m* is defined by the matrix:

$$m = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x & y & 1 \end{matrix}$$

<*a*: Number> ROTATE → <*m*: Transformation>

where the effect is $cos(a)\ -sin(a)\ 0\ sin(a)\ cos(a)\ 0$ MAKET. The angle *a* is measured in degrees. The rotation transformation can be viewed in two ways: it rotates coordinate axes *clockwise* by the angle *a*; or it rotates geometrical figures *counterclockwise* by the angle *a*. The transformation *m* is defined by the matrix:

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

<*s*: Number> SCALE → <*m*: Transformation>

where the effect is *s* 0 0 0 *s* 0 MAKET. The transformation *m* is defined by the matrix:

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

<*sx*: Number> <*sy*: Number> SCALE2 → <*m*: Transformation>

where the effect is *sx* 0 0 0 *sy* 0 MAKET. The transformation *m* is defined by the matrix:

$$\mathrm{m} = \begin{matrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{matrix}$$

The image can be reflected about the *y* axis by <-1 1 SCALE2>, or about the *x* axis by <1 −1 SCALE2>. If $|sx| \neq |sy|$ the transformation is not orthogonal; SCALE2 and MAKET are the only ways to generate a non-orthogonal transformation.

<*m*: Transformation> <*n*: Transformation> CONCAT → <*p*: Transformation>

where $p = mn$, i.e., it is the composition of the two transformations m and n formed by multiplying the matrices. Small numeric errors may occur with each concatenation; care must be exercised to avoid error accumulation (see §5.6).

4.4.4 Applying transformations

Interpress has no primitives which apply the T_p or T_v function and return a result on the stack. The current transformation is applied automatically to coordinates by mask and current position operators, and several other primitives.

4.4.5 The current transformation

Several imaging operators work in conjunction with the current transformation, which is the value of the imager variable T; since T is a non-persistent imager variable, it is saved and restored by DOSAVE and DOSAVEALL. When the interpretation of an Interpress master begins, T is set to the identity transformation (i.e., <1 SCALE>). The *SETMEDIUM operator (§4.2) alters T to establish the Interpress coordinate system.

The intention is that the Interpress coordinate system, or some more convenient system based on it, be used to describe the entire page, often supplemented by master coordinate systems used within instances that will be related to the Interpress coordinate system by an incremental transformation. For this reason, the operators shown below all make incremental changes to T in order always to incorporate the Interpress-to-device transformation as part of any current transformation.

<*m*: Transformation> CONCATT → < >

where the effect is T IGET CONCAT T ISET; i.e., T is set to <*m* T CONCAT>.

< > MOVE → < >

where the effect is GETCP TRANSLATE CONCATT; i.e., T is modified so that the origin (0, 0) maps to the current position. GETCP and the current position are defined in §4.5.

< > TRANS → < >

where T is modified so that the origin (0, 0) maps to the rounded current position. More precisely, the effect is equivalent to { *DCScpx* IGET *DCScpy* IGET *DROUND *DCScpy* ISET *DCScpx* ISET GETCP } MAKESIMPLECO DOSAVEALL TRANSLATE CONCATT.

The rounding operation in TRANS implies that any coordinates to which T is subsequently applied will be translated by an integral number of grid points. This convention allows often-used instances such as characters to be scan-converted once and then translated at will. TRANS is designed together with SETXYREL (§4.5) to achieve positioning precision, while still letting each instance of a character be scan-converted identically.

4.5 Current position operators

The Interpress imaging operators make it easy to locate a graphical object such as a character at the *current position*, a location on the page image. The current position is measured in the device coordinate system, and is recorded in two persistent imager variables *DCScpx* and *DCScpy*. Several operators are available for changing the current position. It is by altering the current position that an operator displaying a character specifies where the next character on the text line should usually lie.

The operators for changing the current position take arguments in a coordinate system defined by the master and convert them to the device coordinate system using the current transformation T.

<*x*: Number> <*y*: Number> SETXY→ < >
where the current position is set to the coordinate determined by transforming *x* and *y*. Precisely, $(DCScpx, DCScpy) := T_p(x, y, T)$.

<*x*: Number> <*y*: Number> SETXYREL→ < >
where a relative displacement is added to the current position. Precisely, $(DCScpx, DCScpy) := T_v(x, y, T) + (DCScpx, DCScpy)$.

<*x*: Number> SETXREL→ < >
where the effect is *x* 0 SETXYREL; i.e., a relative displacement in the *x* direction is added to the current position.

<*y*: Number> SETYREL→ < >
where the effect is 0 *y* SETXYREL; i.e., a relative displacement in the *y* direction is added to the current position.

< > GETCP→ <*x*: Number> <*y*: Number>
where $T_p(x, y, T) = (DCScpx, DCScpy)$. It is a master error if the matrix T is too poorly conditioned to invert.

4.6 Pixel arrays

Interpress allows both masks and colors to be described by *pixel arrays*, arrays of numeric samples of pixels on a two-dimensional grid. This section explains the conventions behind the *PixelArray* type, which represents these arrays.

A PixelArray is constructed with the following primitive:

<*xPixels*: Cardinal> <*yPixels*: Cardinal> <*samplesPerPixel*: Cardinal>
<*maxSampleValue*: Cardinal or Vector of Cardinal>
<*samplesInterleaved*: Cardinal> <*m*: Transformation>
<*samples*: Vector> MAKEPIXELARRAY→<*pa*: PixelArray>
where the effect is complex, and is explained in the rest of this section.

The definition of a pixel array proceeds in two stages. First, a rectangular array of pixels is defined in the *pixel array coordinate system*. The rectangular array defines an image in the region $0 \le x \le xPixels$, $0 \le y \le yPixels$. Each pixel is defined by *samplesPerPixel* separate samples, denoted by $s_0, s_1, \ldots, s_{samplesPerPixel-1}$. If *maxSampleValue* is a Cardinal, all sample values lie between 0 and *maxSampleValue* inclusive; otherwise, *maxSampleValue* is a Vector that gives the maximum sample value for each sample. More precisely, each sample value s_i is a Cardinal in the range $0 \le s_i \le$ (**if** <*maxSampleValue* TYPE> = 1 **then** *maxSampleValue* **else** <*maxSampleValue* *i* GET>). A master error occurs if *maxSampleValue* is a Vector and <*maxSampleValue* SHAPE> is not <0 *samplesPerPixel*>. The interpretation of sample values depends on how the PixelArray is used; it is described in §4.7 for sampled color, and in §4.8.4 for masks.

A pixel said to be *located* at (p_x, p_y) describes a region of the image centered about the point $(p_x + \frac{1}{2}, p_y + \frac{1}{2})$, and extending a distance slightly more than $\frac{1}{2}$ in all directions. The pixel intensity profile is not defined in detail, but may be assumed to be roughly as shown in Figure 4.3.

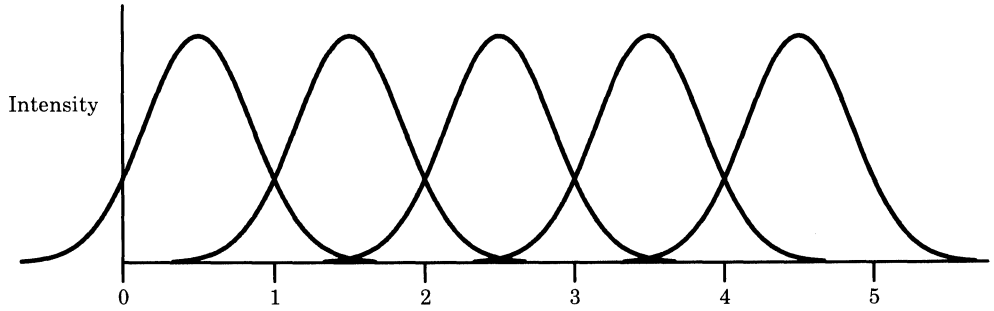


Figure 4.3 **Pixel intensity profiles**

The *samples* vector must contain $xPixels \times yPixels \times samplesPerPixel$ samples, each recorded in a separate element in the vector. The pixel located at (x, y) is defined by samples $s_0, s_1, \ldots, s_{samplesPerPixel-1}$, where $s_i = <samples\ j$ GET> and $j =$ (**if** *samplesInterleaved* **then** $(x \times yPixels + y) \times samplesPerPixel + i$ **else** $xPixels \times yPixels \times i + (x \times yPixels + y)$).

Informally, the sequence of samples in the vector is such that a rectangular grid is scanned out in a series of scan-lines. The first pixel in the vector is located at (0, 0), the next pixel at (0, 1), and so forth up to (0, $yPixels - 1$), defining the first scan-line . Then the next scan-line is described: the next pixel is at (1, 0), followed by (1, 1) up to (1, $yPixels - 1$). The final scan-line defines pixels at locations ($xPixels - 1.0$) to ($xPixels - 1, yPixels - 1$).

If *samplesInterleaved* is non-zero, all *samplesPerPixel* samples for a pixel are located together in the samples vector. Thus the first *samplesPerPixel* samples in the vector are the samples for the pixel located at (0,0). If *samplesInterleaved* is zero, then the samples vector is conceptually divided into *samplesPerPixel* separate sequences, each containing $xPixels \times yPixels$ samples. The first sequence describes the values of the first sample over the entire grid, the second the values of the second sample, etc.

The second stage in the definition of a PixelArray is a coordinate transformation, which describes how to transform pixel locations in the pixel array coordinate system into positions that will appear meaningful when printed. The intent of the transformation is to capture different scanning orders under which the *samples* vector may have been recorded. The normal convention is that this transformation converts the rectangle defined in the pixel array coordinate system into a new rectangle with the (0, 0) point of the rectangle at the lower-left corner when the image is "upright," the y axis pointing up and ranging from 0 to some positive value, and the x axis pointing to the right and ranging from 0 to some positive value.

If, for example, an image were scanned using vertical scan-lines scanned bottom-to-top with scan lines appearing in left-to-right order when the image is held upright, then the transformation might be <1 SCALE>, the identity. If the scan-lines are scanned top-to-bottom, the transformation would be <1 −1 SCALE2 0 *yPixels* TRANSLATE CONCAT>. If the image is scanned using horizontal scan-lines scanned left-to-right with scan lines appearing in top-to-bottom order when the image is held upright, then the transformation would be <−90 ROTATE 0 *xPixels* TRANSLATE CONCAT>.

It is the intention of the standard that a scanned image retained by a printer to be used as a form will be recorded in the file system using these conventions. A master will then incorporate the PixelArray definition from the file system using the *sequenceInsertFile* encoding notation (§2.5.3) and pass it on to a mask operator.

When a pixel array is used to define a mask (§4.8.4) or sampled color (§4.7), another transformation *um* is supplied which maps the pixel array to device coordinates. Because *um* must map to device coordinates, it will usually be T or some transformation derived from T. For a mask, *um* is T when MASKPIXEL is executed; for a sampled color, it is the *um* argument to MAKESAMPLEDBLACK or MAKESAMPLEDCOLOR. The *net* transformation for the pixel array <*xPixels yPixels* 1 1 1 *m samples* MAKEPIXELARRAY> is $nm = <m\ um$ CONCAT T_{ID}^{-1} CONCAT>, where T_{ID}^{-1} is the inverse of the ICS-to-DCS transformation (§4.3.4); nm takes the pixel array coordinate system into the Interpress coordinate system. Some printers may

specify a set of *easy* values of *nm* which they handle efficiently, and may be unable to handle a pixel array at all unless *nm* is easy (§5.4.2).

The EXTRACTPIXELARRAY operator selects certain samples from a pixel array and constructs a new pixel array:

<*p*: PixelArray> <*select*: Vector of Cardinal> EXTRACTPIXELARRAY → <*pa*: PixelArray>
where *pa* is formed from *p* by extracting certain samples from every pixel. The properties *xPixels*, *yPixels*, and *m* are the same for *pa* and for *p*. The Vector *select* gives the indices, in the range from 0 to *p*'s *samplesPerPixel* − 1 inclusive, of the samples to include in *pa*. For each pixel in *pa*, the value of a sample s_i is the same as the value of sample s_j for the corresponding pixel in *p*, where i = <*select j* GET>. The value of *samplesPerPixel* for *pa* will be *n*, the number of elements in the Vector *select*, i.e., n = <*select* SHAPE EXCH POP>. The maximum sample values for a sample in *pa* will be the same as for the corresponding sample in *p*.

Suppose, for example, that a pixel array stored in frame element 1 has four samples per pixel. Samples 0, 1, and 2 are part of a three-color image, while sample 3 is a binary value used as a mask. Since Interpress requires that the color and mask be used separately, EXTRACTPIXELARRAY is used to separate them from the original pixel array. The sequence <1 FGET [0 1 2] EXTRACTPIXELARRAY> yields the three-color pixel array, while <1 FGET [3] EXTRACTPIXELARRAY> obtains the mask.

4.6.1 Compressing sample vectors

When an Interpress master includes a large pixel array, its *samples* vector is often compressed. The compressed samples are expressed as a single vector, plus an operator that can "decompress" the data in the vector into the expanded form of the *samples* vector. The decompression operator is obtained from the environment, then called with some form of DO; the following illustrates the form of a decompression operator:

<*v*: Vector> *decompress* DO → <*samples*: Vector>
where *v* contains the compressed pixel data and any additional parameters the *decompress* operator may need.

A decompression operator is obtained by the FINDDECOMPRESSOR primitive:

<*v*: Vector> FINDDECOMPRESSOR → <*o*: Operator>
where *v* is a Vector of Identifiers which is the universal name of a decompression operator. The operator is returned as *o*. If *o* is applied to a vector containing pixel data compressed in the proper way, it returns the uncompressed vector.

Note that decompression operators are intended to be used only in making PixelArrays: e.g.,

```
300 600 1 1 1
1 scale
[-- compressed pixel vector --] [Xerox, packed] FINDDECOMPRESSOR DO
MAKEPIXELARRAY.
```

If they are executed in other contexts, limits of the implementation may be exceeded or poor performance may result.

Note that decompression operators may reorder data arbitrarily so as to conform to the format of the *samples* vector required by MAKEPIXELARRAY. Also, decompression operators can be defined that convert non-rectangular scanning regimes (e.g., hexagonal) into samples on a rectangular grid.

4.7 Color

The color that will be deposited on the page image is determined by the value of the *color* variable when a mask operator is invoked. Wherever the mask allows it, the color specified by the imager variable *color* is deposited on the page image, obliterating (or modifying) any color previously laid down at the same position on the page. There are two ways to specify the color that will be deposited on the page image where the mask allows: a constant color, and color sampled on a raster. A value of type *Color* fully specifies a color; a subtype *ConstantColor* is used for constant colors.

The *color* variable, which determines the color deposited on the page by a mask operator, is initialized to <1 MAKEGRAY>, or full black. It can be set with ISET. There are also convenience operators for setting it to a constant gray or a sampled color.

4.7.1 Constant color

A *constant color* deposits the same color at each point of the mask. Constant colors may be obtained with MAKEGRAY or FINDCOLOR and used to set the *color* imager variable:

<*f*: Number> MAKEGRAY → <*col*: ConstantColor>
where *col* represents a shade of gray specified by f, $0 \leq f \leq 1$. Informally, $f=0$ corresponds to white, and $f=1$ to black. More formally, $f=0$ corresponds to the maximum intensity, and $f=1$ to the minimum intensity achievable with the imaging medium under normal viewing conditions. Intermediate values of f produce intermediate intensities: $I_f = I_0 - f(I_0 - I_1)$, where I_f is the intensity of light energy corresponding to f.

MAKEGRAY is intended to yield neutral colors linearly spaced in the range of intensities achievable with the imaging medium. For a printer depositing black ink on white paper, $f=0$ corresponds to the paper color, and $f=1$ to the color of black ink on the paper; $f=\frac{1}{2}$ might be implemented with a checkerboard pattern of ink. For a negative image on film, $f=0$ corresponds to minimum transmittance, and $f=1$ to maximum transmittance. For a video display, $f=0$ corresponds to maximum emitted intensity, and $f=1$ to minimum intensity.

<*v*: Vector> FINDCOLOR → <*col*: ConstantColor>
where *v* is a Vector of Identifiers which is the universal name of the desired color. If the specified color cannot be found, an appearance error is generated, and an approximation to the color is returned. Examples of color names might be *Xerox/highlight* or *nbs/cns/bluegreen*.

Sometimes rather than naming each color separately, it is useful to parameterize colors using a color coordinate system. A *color operator* is an Operator that behaves in the following way:

<*coords*: Vector of Number> *colorOperator* DO → <*col*: ConstantColor>
where *col* represents the color whose coordinates are *coords* in the color coordinate system used by *colorOperator*. For example, a *colorOperator* might be designed to interpret measurements in the CIE system, in which case *coords* would be a three-element vector giving the X, Y, and Z coordinates of the desired color.

Normally, color operators will not be specified in a master, but will rather be obtained from the environment with the following operator:

<*v*: Vector> FINDCOLOROPERATOR → <*colorOperator*: Operator>
where *v* is a Vector of Identifiers which is the universal name of the desired color operator.

Sometimes a color operator is part of a family of color operators that all have certain similarities. A *color model operator* is an Operator that, when called, returns a color operator. It behaves in the following way:

<*parameters*: Vector> *colorModelOperator* DO → <*colorOperator*: Operator>
where *parameters* is a Vector that determines exactly what kind of *colorOperator* is desired.

For example, consider a class of operators that accept gray-scale pixel arrays, in which each pixel is defined by a single sample in the range $s_{white}, \ldots, s_{black}$. These operators scale each sample so that a sample whose value is s_{white} will correspond to paper color and a sample whose value is s_{black} will correspond to the blackest ink achievable, and samples in between will scale linearly between white and black. All of these operators can be described by a single color model operator, which takes the argument $[s_{white}, s_{black}]$ and returns a color operator, which in turn will map an argument $[s_0]$ to the appropriate shade of gray, equivalent to <*f* SETGRAY>, where $f = min(max((s_0 - s_{white})/(s_{black} - s_{white}), 0), 1)$.

Color model operators are usually obtained from the environment with the following operator:

<*v*: Vector> FINDCOLORMODELOPERATOR → <*colorModelOperator*: Operator>
where *v* is a Vector of Identifiers which is the universal name of the desired color model operator.

4.7.2 Sampled color

A *sampled color* allows the presentation of stipple patterns, photographs, and other images with rapidly varying colors. The idea is to specify the color at each point in a two-dimensional array; the array is then transformed to appear on the page at an arbitrary position. Large areas can be *tiled* by repeating the sample array.

<*pa*: PixelArray> <*um*: Transformation> <*colorOperator*: Operator>
MAKESAMPLEDCOLOR → <*col*: Color>
where *pa* provides color samples, *um* is a transformation that maps the region defined by *pa* to device coordinates, and *colorOperator* is an operator that maps each pixel's samples into the appropriate color.

If *pa* is made by <*xPixels yPixels samplesPerPixel maxSampleValue samplesInterleaved pm samples* MAKEPIXELARRAY>, then <*pa um colorOperator* MAKESAMPLEDCOLOR> defines a region in the Interpress coordinate system which is the rectangle with corners at (0, 0) and (*xPixels*, *yPixels*), transformed by the net transformation $nm = <pm\ um$ CONCAT T_{ID}^{-1} CONCAT> (§4.6). This region is used as a *tile* to build an arbitrarily large pattern of color which encompasses the entire page image—see Figure 4.4.

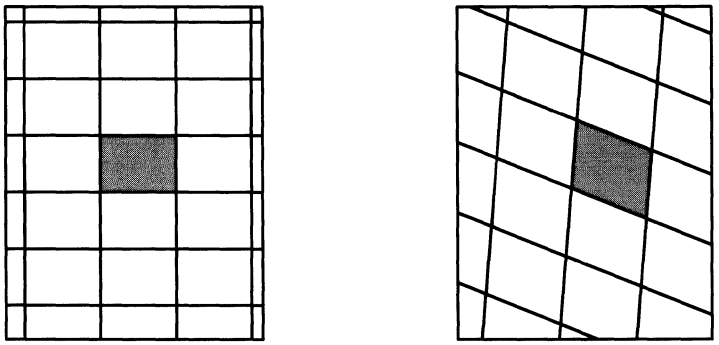
The color of a pixel in *pa* with sample values $s_0, s_1, \ldots, s_{samplesPerPixel-1}$ is obtained by

<$[s_0, s_1, \ldots, s_{samplesPerPixel-1}]$ 1 MARK *colorOperator* DOSAVEALL 1 UNMARK>.

Note that execution of the color operator cannot have any side effects.

It is not expected that an Interpress implementation will actually call *colorOperator* once for every pixel in the pixel array. If *colorOperator* is obtained with FINDCOLOROPERATOR or FINDCOLORMODELOPERATOR, the imager probably has a highly efficient implementation of the color computation, perhaps even performed in special hardware. Creators should in general not compose their own *colorOperators* to use with MAKESAMPLEDCOLOR, because the performance penalty may be intolerable.

To fill in an outline with some repetitive pattern, such as a cross-hatch, find the smallest image which can be replicated by tiling to produce the pattern. Construct a PixelArray which specifies this image, apply MAKESAMPLEDCOLOR to obtain a color, and store it into the *color* variable. Then construct the desired outline, and use MASKFILL to apply the pattern to the region defined by the outline.



Normally the tile is a rectangle, as in the left-hand picture. The right-hand picture illustrates a pathological case of non-rectangular tiles. The darker parallelogram in both pictures is the one specified by the PixelArray.

Figure 4.4 **Tiling the page with a color parallelogram**

A special form of sampled color, limited to binary black-and-white images, is constructed with the following operator:

<*pa*: PixelArray> <*um*: Transformation> <*clear*: Cardinal> MAKESAMPLEDBLACK → <*col*: Color>

where *pa* provides binary samples, *um* is a transformation that maps the region defined by *pa* to device coordinates, and *clear* is 0 or 1. The *samplesPerPixel*, *maxSampleValue*, and *samplesInterleaved* parameters of *pa* must all be 1. In all respects except the determination of the color to deposit, the effect of MAKESAMPLEDBLACK is like that of MAKESAMPLEDCOLOR.

The color deposited through the mask on an image pixel corresponding in position to a pixel array sample value of x depends on x. If $x=1$, the color is black, i.e., <1 MAKEGRAY>. If $x=0$, the effect depends on *clear*: for *clear*=0 the color is white, i.e., <0 MAKEGRAY>; for *clear*=1 no ink is deposited, i.e., the color is clear.

4.7.3 Convenience operators

The following convenience operator sets the color variable to a constant gray:

<*f*: Number> SETGRAY → < >

where the effect is *f* MAKEGRAY *color* ISET.

The following convenience operators set the color variable to a sampled color:

<*pa*: PixelArray> <*m*: Transformation> <*colorOperator*: Operator> SETSAMPLEDCOLOR → < >

where the effect is *pa m T* IGET CONCAT *colorOperator* MAKESAMPLEDCOLOR *color* ISET.

<*pa*: PixelArray> <*m*: Transformation> <*clear*: Cardinal> SETSAMPLEDBLACK → < >

where the effect is *pa m T* IGET CONCAT *clear* MAKESAMPLEDBLACK *color* ISET.

Note that these operators concatenate the current transformation *T* to their transformation argument before calling MAKESAMPLEDCOLOR or MAKESAMPLEDBLACK.

4.8 Mask operators

The mask operators are the central focus of the Interpress master, for they determine the shapes of images that are laid down on the page image. The most common shapes are those used to make images of characters; these masks are specified in pre-defined sets of character definitions called *fonts* (§4.9). Mask operators are also available to make images of rectangles, line drawings, or filled outlines, and to use a pixel array to specify samples of the mask.

When a mask operator is executed, the page image is altered. The operation of a mask operator is controlled in part by its arguments and in part by imager variables. The following variables affect all mask operators:

- *T*, the current transformation. The mask commands all require sizes and coordinates, which will be transformed by the current transformation to determine the coordinates of the mask on the page image.

- *color*. The *color* variable governs the color of the object that will be placed on the page image by a mask operator (§4.7).

- *priorityImportant*. The priority order of objects laid down when *priorityImportant*$\neq 0$ is preserved (§4.1.1).

- *noImage*. If *noImage* is non-zero, a mask operator will have no effect on the page image, although it will have the proper effect on the stack and imager variables. If *noImage* is zero, the operator will alter the image as explained below. When the interpretation of an Interpress master begins, *noImage* is set to zero. The purpose of *noImage* is explained in §4.10.

- *clipper*. The *clipper* variable defines the region on the page in which mask operators will be able to alter the page image. Modifications to the page image that lie outside this region are inhibited (§4.8.5).

4.8.1 Geometry: trajectories and outlines

Shapes are defined geometrically in terms of *segments*, *trajectories*, and *outlines*. A *segment* is a directed line or curve segment; it has a *start point* and an *end point*. A *trajectory* is a sequence of connected segments; the end point of a segment coincides with the start point of the next one. A *closed trajectory* is a trajectory that closes upon itself, that is, the end point of the last segment in the trajectory coincides with the start point of the first segment. An *outline* is a collection of trajectories; each trajectory in an outline is implicitly closed by a straight-line segment linking the end point of the last segment with the start point of the first segment.

Trajectories and outlines are represented by two corresponding Interpress types. Values of these types are data structures that are built by constructor operators described in this section. There are no operators for decomposing trajectories or outlines into their constituent parts, because values of these types are used simply as a way to pass a description of a complex shape to an imaging operator.

Trajectories and outlines are given a geometrical interpretation only when they are used to specify a mask or clipping outline. At this point, the numbers describing the trajectory or

outline are interpreted as defining geometry in the master coordinate system; imaging operators such as MASKFILL and MASKSTROKE convert this geometry to the device coordinate system by applying the current transformation T. Thus the value of T while the trajectory is constructed is ignored; only the value of T when the mask operator is executed is important.

Trajectories may be constructed with primitive operators. A trajectory is started with MOVETO, which places on the stack a *trajectory* value describing the trajectory. Then the trajectory is extended by operators such as LINETO or CURVETO, which add a segment to a trajectory. A *last point* (*lp*) is always associated with a trajectory: it is the end point of the last segment in the trajectory.

<x: Number> <y: Number> MOVETO → <t: Trajectory>
where t describes a new trajectory; t's *lp* is (x, y).

<t_1: Trajectory> <x: Number> <y: Number> LINETO → <t_2: Trajectory>
where t_2 is formed by extending t_1 with a straight-line segment from t_1's *lp* to the point (x, y); t_2's *lp* is (x, y).

<t_1: Trajectory> <x: Number> LINETOX → <t_2: Trajectory>
where the effect is <t_1 x y_0 LINETO>, where (x_0, y_0) is t_1's *lp*.

<t_1: Trajectory> <y: Number> LINETOY → <t_2: Trajectory>
where the effect is <t_1 x_0 y LINETO>, where (x_0, y_0) is t_1's *lp*.

<t_1: Trajectory> <x_1: Number> <y_1: Number> <x_2: Number> <y_2: Number>
<x_3: Number> <y_3: Number> CURVETO → <t_2: Trajectory>
where t_2 is formed by extending t_1 with a cubic curve segment. Let $P_0=(x_0, y_0)=t_1$'s *lp*, $P_1=(x_1, y_1)$, $P_2=(x_2, y_2)$, and $P_3=(x_3, y_3)$. The curve is determined by Bézier control points P_0, P_1, P_2, P_3 in order; it starts at P_0, tangent to and pointed in the direction of the line from P_0 to P_1; it ends at P_3, tangent to and pointed in the direction of the line from P_2 to P_3; t_2's *lp* is P_3.

The Bézier control points represent the parametric curve

$$x = a_x t^3 + b_x t^2 + c_x t + d_x,\ y = a_y t^3 + b_y t^2 + c_y t + d_y,\ 0 \le t \le 1, \text{ where}$$

$$a_x = x_3 - 3x_2 + 3x_1 - x_0,\ b_x = 3x_2 - 6x_1 + 3x_0,\ c_x = 3x_1 - 3x_0,\ d_x = x_0,$$
$$a_y = y_3 - 3y_2 + 3y_1 - y_0,\ b_y = 3y_2 - 6y_1 + 3y_0,\ c_y = 3y_1 - 3y_0,\ d_y = y_0.$$

Bézier curves are extremely versatile: any parametric cubic curve can be expressed as a Bézier curve; a simple computation converts coefficients of the parametric cubic equations into Bézier control points. A wide range of curves can be generated by fitting several Bézier curves together so as to preserve continuity at the joints. Because the x and y components of a Bézier curve are defined by independent parametric equations, the same curve results whether the curve is drawn in master coordinates and then transformed into device coordinates, or the control points are first transformed to device coordinates and then used to draw the Bézier curve defined by them.

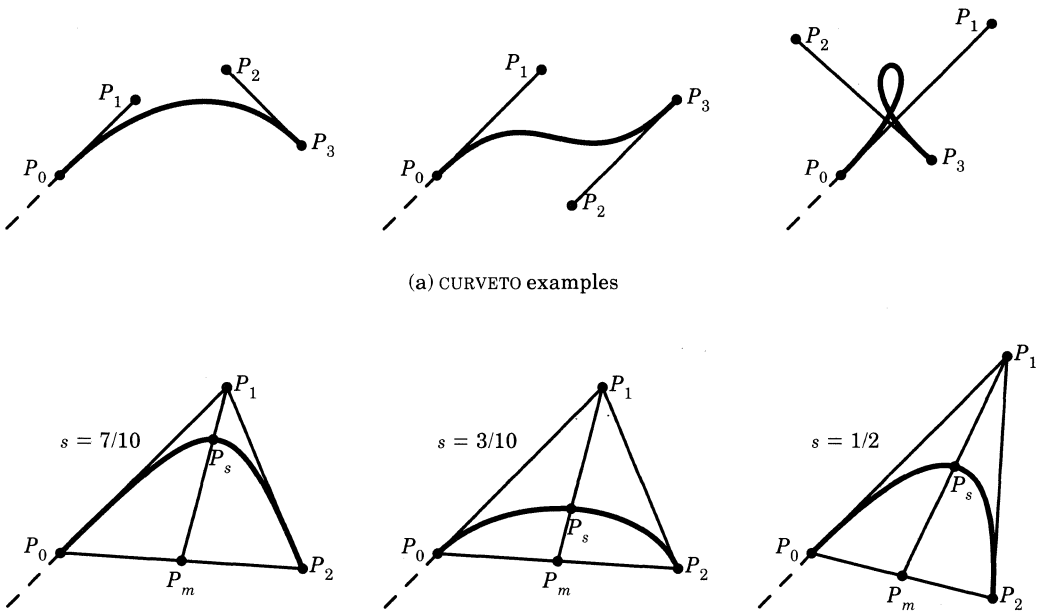
<t_1: Trajectory> <x_1: Number> <y_1: Number> <x_2: Number> <y_2: Number>
<s: Number> CONICTO → <t_2: Trajectory>
where t_2 is formed by extending t_1 with a conic curve segment. Let $P_0=(x_0, y_0)=t_1$'s *lp*, $P_1=(x_1, y_1)$, and $P_2=(x_2, y_2)$. The curve is a piece of a conic section; it starts at P_0, tangent to and pointed in the direction of the line from P_0 to P_1; it ends at P_2, tangent to and pointed in the direction of the line from P_1 to P_2; t_2's *lp* is P_2. The shape parameter s, $0 \le s \le 1$, is the ratio of the distances $P_m P_s$ and $P_m P_1$, where P_m is the midpoint of $P_0 P_2$, and P_s is the point at which the curve intersects $P_m P_1$.

The segment is a piece of an ellipse if $s<\frac{1}{2}$, a piece of a parabola if $s=\frac{1}{2}$, and a piece of a hyperbola if $s>\frac{1}{2}$. Note that three or more CONICTO calls are required to construct a closed trajectory such as an ellipse or circle. As with Bézier curves, the same curve results whether the curve is drawn in master coordinates and then transformed into device coordinates, or the control points are first transformed to device coordinates (leaving s unchanged) and then used to draw the conic section defined by them.

The convenience operator ARCTO appends a circular arc to a trajectory; the effect of the operator is equivalent to one or more invocations of CONICTO. Note that non-uniform scaling in the current transformation may distort circles defined in master coordinates into ellipses on the page image.

$<t_1$: Trajectory$>$ $<x_1$: Number$>$ $<y_1$: Number$>$ $<x_2$: Number$>$ $<y_2$: Number$>$
ARCTO → $<t_2$: Trajectory$>$

where t_2 is formed by extending t_1 with an arc of a circle. Let $P_0=(x_0, y_0)=t_1$'s *lp*, $P_1=(x_1, y_1)$, and $P_2=(x_2, y_2)$; the arc starts at P_0, passes through P_1, and ends at P_2; t_2's *lp* is P_2. It is recommended that P_1 lie near the halfway point along the arc. If P_2 is coincident with P_0, the arc is a counterclockwise full circle with diameter P_0P_2. Otherwise, if the three points are collinear, the effect is equivalent to $<x_1\ y_1$ LINETO $x_2\ y_2$ LINETO$>$.



(a) CURVETO examples

(b) CONICTO examples

(c) ARCTO examples

Figure 4.5 **Curved trajectory segments**

An *outline* is represented by a separate type, built using the one of the following operators:

$<t_1$: Trajectory$>$ $<t_2$: Trajectory$>$ . . . $<t_n$: Trajectory$>$ $<n$: Cardinal$>$ MAKEOUTLINE
→ $<o$: Outline$>$

where the trajectories $t_1, t_2, \ldots, t_n$ together form an outline. Each of the trajectories will be closed if necessary. The outline comprises all points with non-zero winding number.

<t_1: Trajectory> <t_2: Trajectory> . . . <t_n: Trajectory> <n: Cardinal> MAKEOUTLINEODD → <o: Outline>

where the effect is like that of MAKEOUTLINE, except that the outline comprises all points with odd winding number.

Primitives that take an Outline argument need to decide which points lie "inside" the outline. To decide if a point lies inside an outline, it is necessary to compute the point's *winding number*. The winding number counts the number of times the point is surrounded by an outline. The winding number of a point with respect to an outline o is the net number of times a point traversing the (closed) trajectories which form the outline wraps around the given point in a counterclockwise direction. MAKEOUTLINE uses the convention that points with a non-zero winding number lie inside the outline.

Figure 4.6 illustrates several outlines and shows their "insides" according to these two conventions. Note that for multi-trajectory outlines, the order in which points on a trajectory are specified is important.

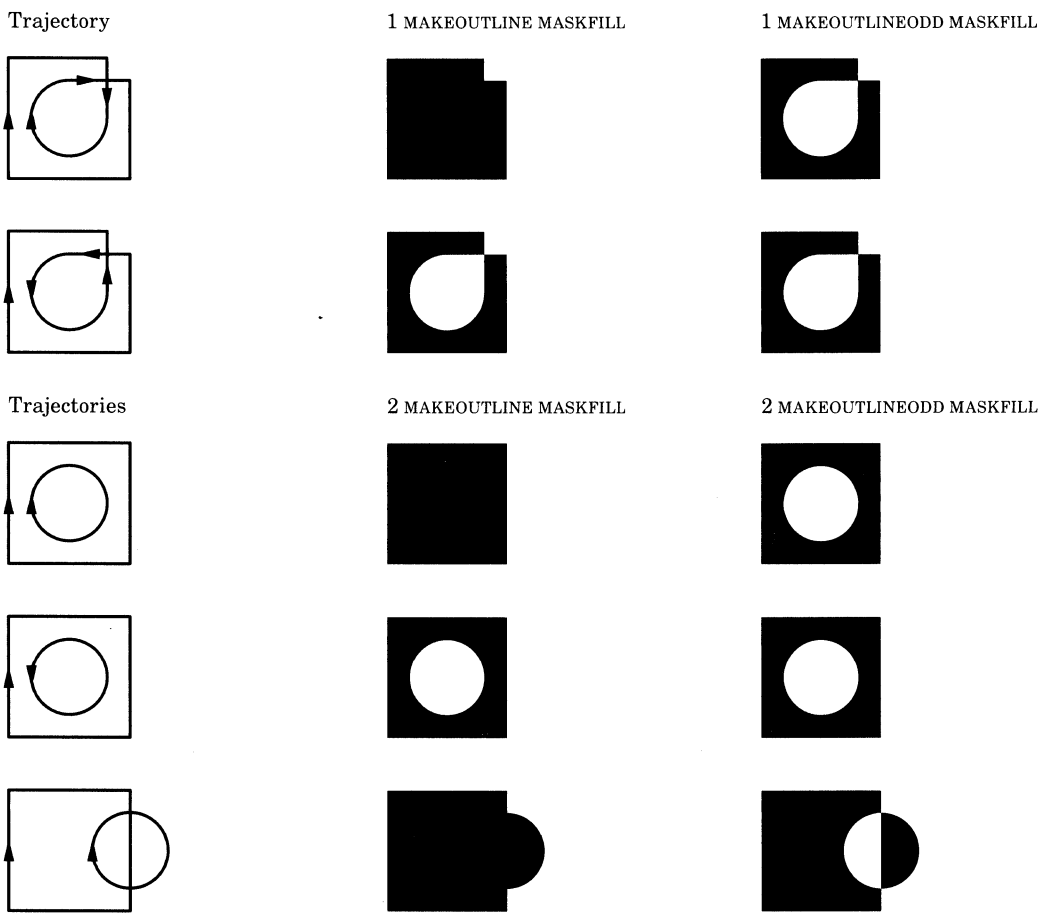


Figure 4.6 **Winding number conventions**

4.8.2 Filled outlines

There is one operator for creating a mask from an outline. It defines a mask to be the "inside" of an outline:

<*o*: Outline> MASKFILL→ < >
where the mask is defined as the region inside the outline *o'*, where *o'* is obtained by transforming *o* into device coordinates using the current transformation *T*.

There is a specialized variant of MASKFILL for imaging an arbitrary rectangle with its sides parallel to the coordinate axes:

<*x*: Number> <*y*: Number> <*w*: Number> <*h*: Number> MASKRECTANGLE→ < >
where the effect is
x y MOVETO *x w* ADD LINETOX *y h* ADD LINETOY *x* LINETOX
1 MAKEOUTLINE MASKFILL
i.e., the mask is a rectangle of width *w* and height *h* with corners at (x, y), $(x+w, y)$, $(x, y+h)$, and $(x+w, y+h)$.

Note that the coordinates of the corners are first computed and then transformed to device coordinates using the current value of *T*; for this reason, the mask on the page image may not be rectangular.

Character strings can be underlined by placing a rectangle of appropriate width and height just below the string. The width of the rectangle will be determined by the width of the character string. The position of the underline along the baseline will be determined by the current position, but because of possible spacing corrections the current position cannot be anticipated accurately when the master is created. The operators STARTUNDERLINE and MASKUNDERLINE are provided to help position underlines accurately. They assume a master coordinate system in which the baseline is oriented in the positive *x* direction.

< > STARTUNDERLINE→< >
where the effect is GETCP POP *underlineStart* ISET; i.e., the *x* component of the current position is remembered as the starting point for an underline.

<*dy*: Number> <*h*: Number> MASKUNDERLINE→< >
where the effect is

```
{2 FSET 1 FSET                    -- now dy=<1 FGET>, h=<2 FGET> --
 GETCP 4 FSET 3 FSET              -- current position X=<3 FGET>, Y=<4 FGET> --
   underlineStart IGET            -- underlineStart --
   4 FGET 1 FGET SUB 2 FGET SUB   -- Y–dy–h --
   SETXY TRANS 0 0                -- set origin to (underlineStart, Y–dy–h) --
   3 FGET underlineStart IGET SUB -- X–underlineStart --
   2 FGET MASKRECTANGLE           -- h --
} MAKESIMPLECO DOSAVEALL          -- don't clobber the frame, current position, or T --
```

That is, the text starting at the point previously identified by STARTUNDERLINE and ending at the current position will be underlined with a rectangle of height *h* and top a distance *dy* below the current position. For example, to underline the word Hello, the master might use STARTUNDERLINE <Hello> SHOW 4 1 MASKUNDERLINE.

The following two convenience operators are provided to specify masks that are filled trapezoids aligned with the coordinate axes:

<x_1: Number> <y_1: Number> <x_2: Number> <x_3: Number> <y_3: Number>
<x_4: Number> MASKTRAPEZOIDX→ < >
where the effect is x_1 y_1 MOVETO x_2 LINETOX x_3 y_3 LINETO x_4 LINETOX 1 MAKEOUTLINE MASKFILL.

<x_1: Number> <y_1: Number> <y_2: Number> <x_3: Number> <y_3: Number>
<y_4: Number> MASKTRAPEZOIDY → < >

where the effect is x_1 y_1 MOVETO y_2 LINETOY x_3 y_3 LINETO y_4 LINETOY 1 MAKEOUTLINE MASKFILL.

4.8.3 Strokes

Several mask operators create a mask from a trajectory which defines the center-line of a stroke to be drawn on the page image. The imager variables *strokeWidth*, *strokeJoint*, and *strokeEnd* control the width of strokes, the shape of joints between stroke segments, and the shape of stroke ends.

<*t*: Trajectory> MASKSTROKE → < >

where the mask is a stroke along trajectory *t*. Each segment of *t* is broadened to have uniform width *strokeWidth*; joint features specified by *strokeJoint* are added between segments; endpoint features specified by *strokeEnd* are added at the ends; and the resulting shape is transformed into device coordinates by the current transformation *T*, and used as a mask to alter the page image.

<*t*: Trajectory> MASKSTROKECLOSED → < >

where *t* is first closed with a straight line segment if necessary, and the mask is a stroke along the resulting closed trajectory. Each segment is broadened to have uniform width *strokeWidth*; joint features specified by *strokeJoint* are added between segments; and the resulting shape is transformed into device coordinates by the current transformation *T*, and used as a mask to alter the page image.

The shape of the stroke in the master coordinate system is the union of the broadened trajectory segments, the joint features, and the endpoint features if any. The precise shape of each of these components is described below.

The broadened shape of a trajectory segment *s* is determined as follows: for each point *p* on *s*, construct a line segment of length *strokeWidth*, perpendicular to *s*, with midpoint at *p*. The broadened shape is the union of all such line segments.

A joint feature fills the gap between broadened segments that occurs on the convex side of the joint if the trajectory slope is discontinuous. There are three options:

strokeJoint=0 (*miter*). Sides of the broadened segments are extended by straight lines in the direction of their respective slopes until they meet, and the resulting quadrilateral is filled. Segments meeting at an acute angle will thus generate long, sharp spikes.

strokeJoint=1 (*bevel*). The corners of the gap are joined by a straight line, and the resulting triangle is filled.

strokeJoint=2 (*round*). The corners of the gap are joined by an arc of a circle of diameter *strokeWidth* centered at the joint, and the resulting sector is filled.

An endpoint feature caps each end of a trajectory. There are three options:

strokeEnd=0 (*square*). A butt end is formed after extending the stroke a distance of half its width in the direction in which the trajectory was pointed at its endpoint.

strokeEnd=1 (*butt*). Ends are simply squared off: no endpoint feature is added.

strokeEnd=2 (*round*). The end is capped with a semicircle whose diameter is *strokeWidth* and whose center coincides with the trajectory endpoint.

If *square* end geometry is undetermined because the trajectory is a single point, an appearance error is generated instead of a mask.

Figure 4.7 illustrates various combinations of end and joint treatment.

The convenience operator MASKVECTOR may be used to draw strokes whose trajectories are a single line segment:

<x_1: Number> <y_1: Number> <x_2: Number> <y_2: Number> MASKVECTOR → < >
where the effect is x_1 y_1 MOVETO x_2 y_2 LINETO MASKSTROKE.

The following operator uses a trajectory and a repeating pattern to define a dashed stroke:

<*t*: Trajectory> <*pattern*: Vector of Number> <*offset*: Number> <*length*: Number> MASKDASHEDSTROKE → < >
where trajectory *t* is the centerline of the dashed stroke; *pattern* specifies a repeating dash pattern, *offset* specifies a starting position in the pattern, and *length*, if positive, specifies how much of the pattern is to be mapped onto the total length of *t*. The effect is
t_{0+d} MASKSTROKE t_{2+d} MASKSTROKE t_{4+d} MASKSTROKE ... t_{2e+d} MASKSTROKE
where trajectories $t_0, t_1, t_2, \ldots, t_m$ are defined as follows:
The last point of t_i coincides with the initial point of t_{i+1} ($0 \leq i \leq m$; m defined below).
$t_0, t_1, t_2, \ldots, t_m$ joined together in order are equivalent to the original trajectory *t*.
n = <*pattern* SHAPE EXCH POP> (number of elements in pattern; require $n > 0$).
l = <*pattern* SHAPE POP> (lower bound of *pattern*).
q_i = <*pattern* i l SUB n MOD GET> (elements of pattern, used cyclically; require $q_i \geq 0$).
L = total arc length of trajectory *t*, in master coordinates.
g = 1 if *length* ≤ 0, L/*length* if *length* > 0.
$Q = q_0 + q_1 + q_2 + \ldots + q_{n-1}$ (total length of pattern; require $Q > 0$).
f = *offset* mod $(2Q)$ (in case *offset* is negative or larger than Q; the 2 is in case n is odd).
k = smallest nonnegative integer such that $q_0 + q_1 + q_2 + \ldots + q_k \geq f$.
$d = k \bmod 2$ (d will be 0 if t_0 is a dash, 1 if it is a gap).
m = smallest nonnegative integer such that $q_0 + q_1 + q_2 + \ldots + q_{k+m} \geq f + L/g$.
$e = \lfloor (m-d)/2 \rfloor$ (largest integer such that $2e + d \leq m$).
$p_0 = q_0 + q_1 + q_2 + \ldots + q_k - f$ (if $m > 0$; if $m = 0$ then $p_0 = p_m = L/g$).
$p_i = q_{i+k}$ $(0 < i < m)$.
$p_m = L/g - (p_0 + p_1 + \ldots + p_{m-1})$.
The arc length of t_i in master coordinates is equal to $g \times p_i$.

The elements of *pattern* specify the lengths of the dashes and the gaps between them; the first element of *pattern* corresponds to a dash. The pattern is repeated as often as necessary. Thus *pattern* = [10] produces alternating 10-unit dashes and 10-unit gaps, while *pattern* = [15, 5] produces 15-unit dashes and 5-unit gaps.

The *offset* specifies the position in the pattern that corresponds to the beginning of the trajectory. Consider *pattern* = [10]. If *offset* = 0, the dashed stroke begins with a 10-unit dash. If *offset* = 6, the dashed stroke begins with a 4-unit dash, since the starting point in the pattern is 6 units into the first dash. If *offset* = 12 (or –8), the dashed stroke begins with an 8-unit gap, followed by a 10-unit dash, a 10-unit gap, and so on.

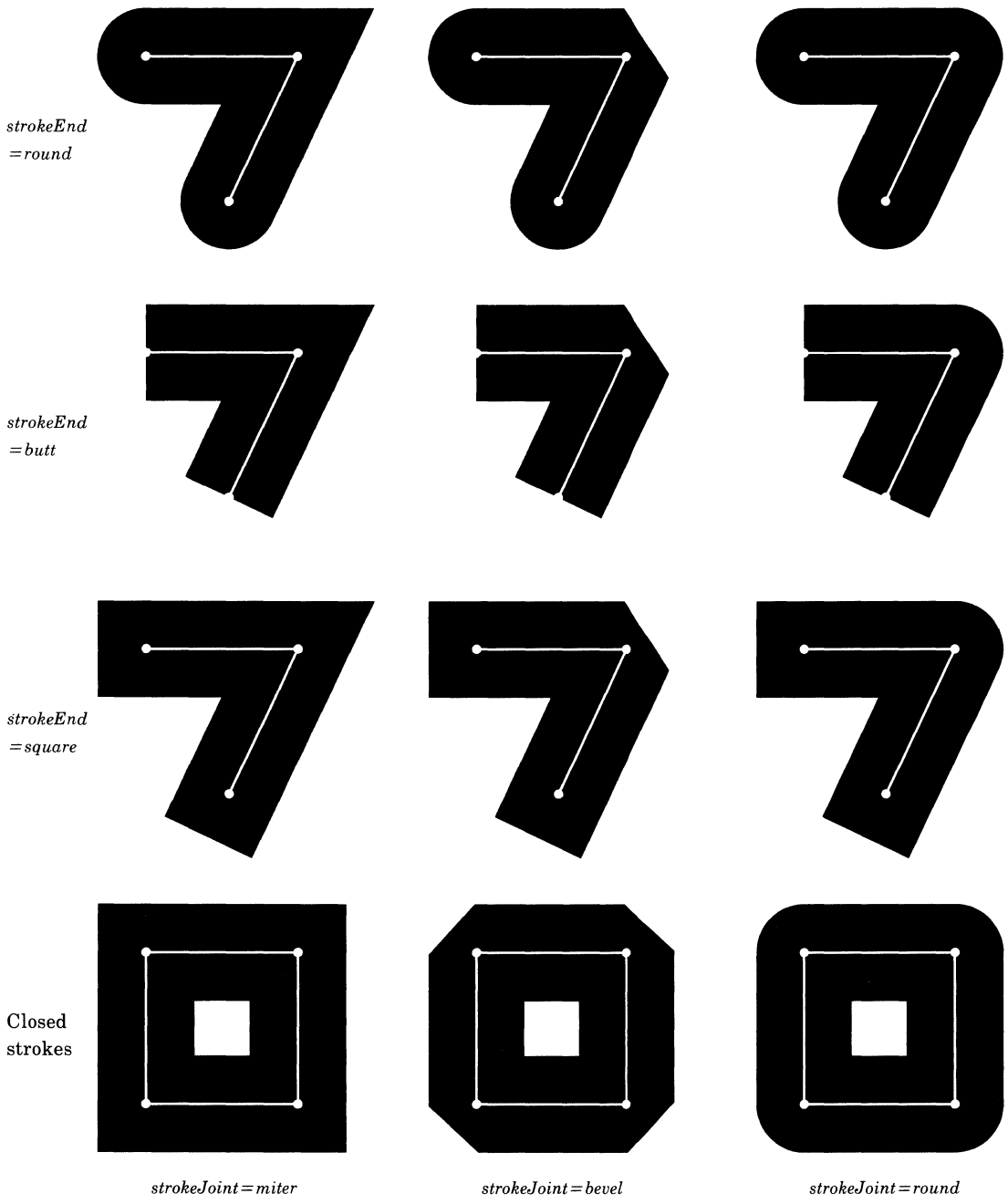


Figure 4.7 **Stroke examples**

The *length*, if positive, specifies how much of the pattern will be stretched or shrunk to fit the entire length of the trajectory. Consider *pattern* = [10] and *offset* = 0, with a trajectory 110 units long. If *length* = 90, 90 units of pattern will be stretched to 110 units in master coordinates; thus there will be exactly 5 dashes and 4 gaps, each 10X(110/90) units long. If *length* = 130, 130 units worth of pattern will be shrunk to 110 units in master coordinates; thus there will be exactly 7 dashes and 6 gaps, each 10X(110/130) units long. If *length* is not positive, it defaults to the actual length of the trajectory in master coordinates.

Figure 4.8 shows several examples of dashed strokes.

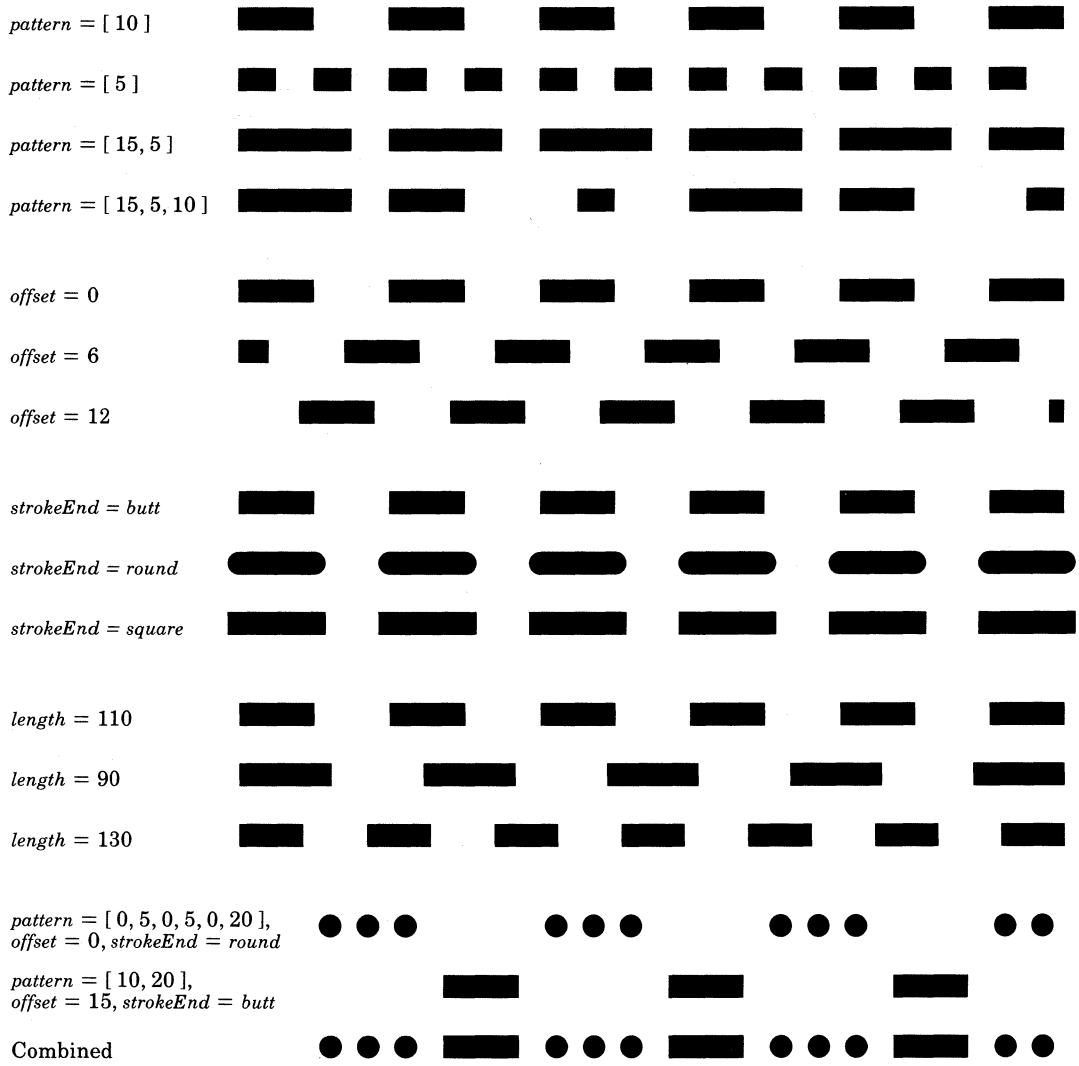


Figure 4.8 **Dashed strokes**

4.8.4 Sampled masks

Some masks are conveniently specified by a two-dimensional array of pixels that describe where the mask lies and where it does not. Such a mask might be obtained by scanning a complicated shape with a raster input scanner.

<*pa*: PixelArray> MASKPIXEL→ <>

where the region defined by *pa*'s pixel array coordinate system is transformed by <*m T* CONCAT> (*m* is the transformation used to make *pa*), thereby defining a region of the page image to be altered. The *samplesPerPixel* and *maxSampleValue* parameters of *pa* must be 1. A sample value of 0 identifies a pixel that is not part of the mask (i.e., where color will not be deposited), while a sample value of 1 identifies a pixel that is completely covered by the mask.

4.8.5 Clipping operators

The region of the page image within which mask operators are allowed to have an effect is called the *clipping region*. The *Clipper* type represents a clipping region; this type is used only by the *clipper* imager variable, which saves the clipping region in device coordinates. When a page body is started, the *SETMEDIUM operator (which cannot be called explicitly by the master) sets the clipping region to a rectangular outline that fills the useable *field* of the page image.

The only operations on the clipping region further restrict it by intersecting the current clipping region with an outline supplied by the master:

<*o*: Outline> CLIPOUTLINE → < >

where outline *o'* is obtained by transforming *o* into device coordinates using the current transformation *T*, then the current clipping region is set to the intersection of the previous clipping region and *o'*. A point will be inside the new clipping region if and only if it lies inside the previous clipping region and inside *o'*.

<*x*: Number> <*y*: Number> <*w*: Number> <*h*: Number> CLIPRECTANGLE → < >

where the effect is

x y MOVETO *x w* ADD LINETOX *y h* ADD LINETOY *x* LINETOX
1 MAKEOUTLINE CLIPOUTLINE.

Note that the coordinates of the corners are first computed and then transformed to device coordinates using the current value of *T*; for this reason, the clipping outline on the page image may not be rectangular.

A creator should do as much clipping as possible as the master is constructed, eliminating objects that lie wholly off the page. Clipping by the printer is required to clip character shapes, which are not accessible to the creator and can therefore not be clipped as the master is constructed.

4.9 Characters and fonts

It is possible to make an image of any character using the mask commands already introduced: pixel arrays and filled outlines are especially well suited to describing character shapes. Unfortunately, an Interpress master that described each character shape each time it was used would be unreasonably long. The master could be shortened considerably by defining a composed operator corresponding to each character; the operator could then be invoked in order to generate a mask of the character. But even a single shape definition of each character would require substantial storage and threaten device-independence.

Instead of requiring that each Interpress master define character shapes in terms of more primitive mask operators, an Interpress printer will generally have a library of character definitions that will provide an operator for each character to be printed. These definitions may even involve device-dependent properties that cannot be specified in an Interpress master itself. For example, a phototypesetter might have optical masters of the characters and a zoom lens to control the size.

Character definitions come in collections called *fonts*. All the characters in a font are designed to appear consistent when printed in words and lines; their widths are chosen so that they juxtapose pleasantly; and they are drawn consistently: their size, style, blackness, and so forth are all compatible.

A font describes the geometry of each character in a *character coordinate system*, in which characters have a standard size and orientation. For each character, there is a corresponding

character operator. Instances of characters are placed on the page by invoking character operators with suitable transformations. 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 next character position*. It alters the current position so as to prepare for the next character in a sequence. Informally, it adds the "width" of the character to the current position. More precisely, it specifies where the origin of the next character should (usually) be placed. This displacement of the current position is called the character's *escapement*.

3. *Corrects spacing*. It may make small adjustments to the current position to compensate for inaccuracies in character escapements. This may happen, for example, if a printer substitutes a different font because it cannot find the requested font in its library.

Generating masks: Figure 4.9 shows examples of character masks defined in the character coordinate system. The masks have an *origin*, shown as (0, 0) in the figures. When a character operator is invoked with SHOW (§4.9.3), an image of the character will be placed on the page so that this origin coincides with the *current position* at the time the character operator is invoked. Consequently, the origin is chosen for convenience in placement. The orientation of the *y* axis is such that it points upward from the origin when the character is viewed in the normal reading orientation.

Moving to the next character position: The character's escapement is represented by the distances *escapementX* and *escapementY*, illustrated in Figure 4.9. After the character's mask is imaged, the current position is altered by executing *escapementX escapementY* SETXYREL.

For most western languages, characters are read horizontally, so normally *escapementY* is zero and *escapementX* is positive. This yields left-to-right spacing as shown in Figure 4.10a. Traditional Chinese characters, which are set top-to-bottom, might have a zero *escapementX* and a negative *escapementY* to establish a current position for a character immediately below the present one; this is illustrated in Figure 4.10b. Hebrew characters, which are set right-to-left, may have negative *escapementX* values, as shown in Figure 4.10c.

When text is justified between fixed margins, the width of "spaceband" characters is adjusted so that the words on the line appear evenly spaced. Such characters, termed *amplifying* characters, achieve the effect of a spaceband by using a slightly different spacing computation, namely *escapementX*amplifySpace escapementY*amplifySpace* SETXYREL, where *amplifySpace* is an imager variable. The width of an amplifying character is determined in part by the font designer, who specifies *escapementX* and *escapementY*, and in part by the master, which sets *amplifySpace*.

Correcting spacing: Character operators work with the CORRECT operator to adjust spacing slightly to compensate for inaccurate character escapements (see §4.10). To provide an opportunity to alter the character spacing slightly, each character operator may call CORRECTSPACE or CORRECTMASK. The type of call is determined by the font designer, but is suggested by the following conventions:

- If the character's width can be adjusted to remedy spacing problems, the operator calls CORRECTSPACE.

- 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 to 9), the operator calls CORRECTMASK.

- A very few character operators may call neither of the correction operators. This will be the case if the spacing after the character must not be altered.

  For example, suppose a font contains a character definition for an acute accent, with *escapementX* = *escapementY* = 0. The accent character operator will be called just before the operator for the character to be accented. The font is designed so that the accent is correctly positioned with respect to the character shape. In this case, even small adjustments to mask positions might make the accented character illegible.

4.9.1 Character coordinate system

The actual size and orientation of a character placed in the image is controlled by the transformation that is current when the character operator is invoked. In order to predict the size of characters on the page, an Interpress creator must know in what units the characters in a font are specified. The character coordinate system serves this purpose.

The character coordinate system for a font in the environment is established by the font designer, according to the universal naming scheme and registry described in §3.2.2. Font designers are strongly encouraged to adopt the following convention whenever possible: in the character coordinate system, characters are defined in portrait orientation, with a scale such that *the body size of the font is one unit.*

The recommended convention follows naturally for fonts, including most typographic fonts, whose sizes are normally given in terms of body height, e.g., in printer's points; thus for a "10 point" font, a distance of 10 points equals one unit in the character coordinate system. For fonts whose sizes are not normally given in terms of body height, e.g., office fonts whose size is usually given in terms of horizontal pitch, some reasonable body size must be chosen; for example, the font Titan 10 pitch might be assigned a body size of 1/6 inch. In all cases, font designers are encouraged to record any non-obvious scaling and orientation information in the registry.

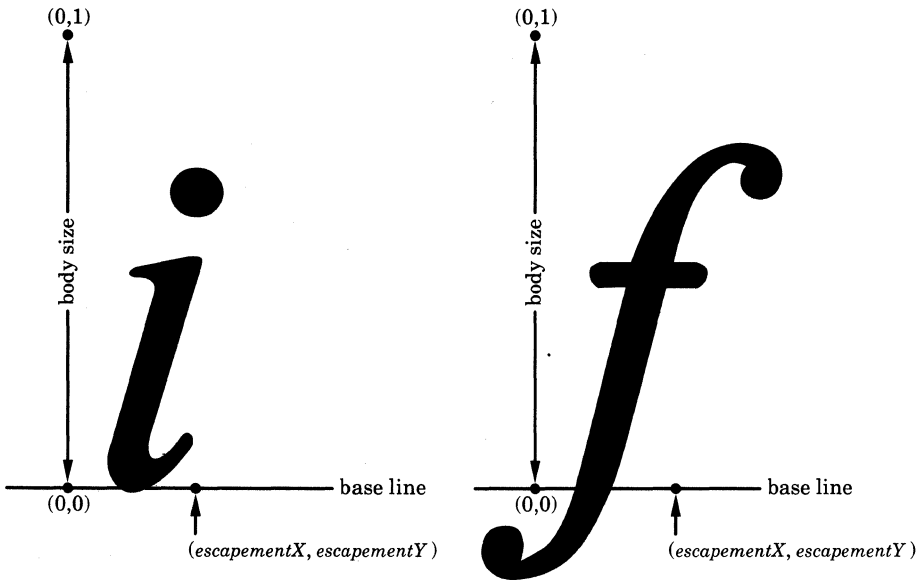
4.9.2 Fonts

The *Font* type represents a font. Fonts are commonly retrieved from a library of fonts in the printer's environment; but an Interpress master may also create a font from character definitions supplied in a vector called a FontDescription. A Font obtained in either of these ways describes its characters in the character coordinate system. A Font may also carry a Transformation that transforms all its characters from the character coordinate system into some other convenient coordinate system.

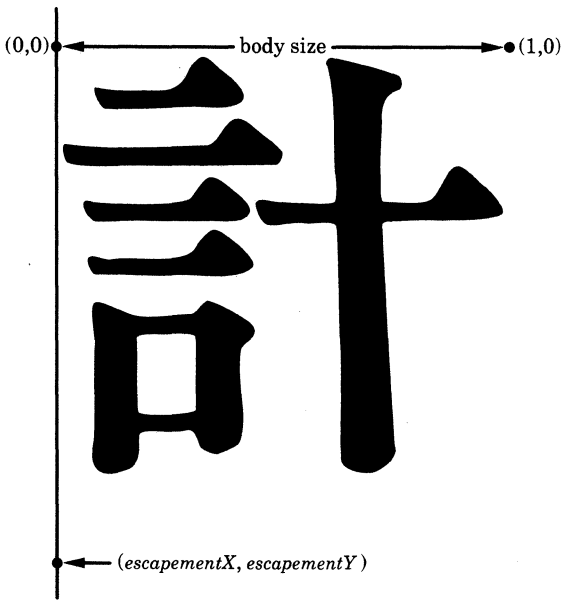
A FontDescription is a universal property vector with the following property names:

transformation: Transformation. Character masks and metrics in the FontDescription may be specified in any coordinate system convenient for the font designer. This transformation converts such FontDescription coordinates into the character coordinate system.

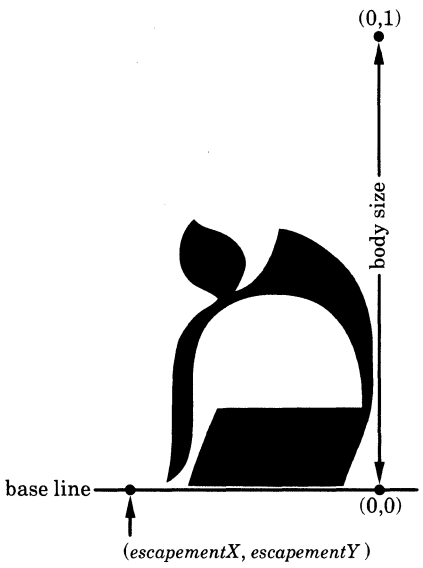
characterMasks: Property vector of character mask operators. Each property name in this vector is a Cardinal, called the *character index*; each corresponding property value is an Operator which generates the character's mask. Interpress establishes no conventions for the correspondence between character shapes and character indices. Thus a font may represent an arbitrary "character set," i.e., an arbitrary mapping from character indices to shapes.



(a) Metrics for left-to-right horizontal spacing



(b) Metrics for top-to-bottom vertical spacing



(c) Metrics for right-to-left horizontal spacing

Figure 4.9 **Character metrics**

In Latin alphabets, such as the italic font shown in Figure 4.9a, the origin is on the *baseline* of a line of characters. For other styles, the origin may be in different locations. The Chinese character in Figure 4.9b has its origin at the upper left; it is intended to be used for setting characters vertically top-to-bottom. The Hebrew character in Figure 4.9c has its origin at the lower right; it is intended to be used for setting characters horizontally right-to-left.

(a) Left-to-right horizontal spacing of English characters

(b) Top-to-bottom vertical spacing of Chinese characters

(c) Right-to-left horizontal spacing of Hebrew characters

Figure 4.10 **Spacing character masks**

characterMetrics: Property vector of CharacterMetrics vectors. Each property name in this vector is a Cardinal character index; each corresponding property value is a CharacterMetrics vector.

substituteIndex: Cardinal. This is the index of a character to be substituted for any character not present in the *characterMetrics* vector.

A CharacterMetrics vector is a universal property vector of metric information about a single character, containing the following properties. Each property specifies a default value to be used if the property is absent.

escapementX: Number. The x-component of the character's escapement. If the *escapementX* property is not present, its value may be assumed to be zero.

escapementY: Number. The y-component of the character's escapement. If the *escapementY* property is not present, its value may be assumed to be zero.

amplified: Cardinal. If this value is nonzero, the character is an amplifying character. If the *amplified* property is not present, its value may be assumed to be zero.

correction: Cardinal. The value indicates what operator, if any, is called to correct spacing: 0=CORRECTMASK, 1=CORRECTSPACE, 2=none. If the *correction* property is not present, its value may be assumed to be zero (i.e., CORRECTMASK).

The FontDescription and CharacterMetrics properties defined above are the only ones used by the imager, and the only ones that need be supplied to MAKEFONT. FontDescription and CharacterMetrics vectors may also include other properties that specify font and character metric information of interest to Interpress creators. Such additional properties are defined by the Xerox Font Interchange Standard.

A Font value is composed of a FontDescription and a Transformation. The FontDescription defines mask operators and metrics in the character coordinate system; the transformation may transform the character coordinate system into another more convenient coordinate system. The following special operators, which cannot be called from the master, assemble a font from a FontDescription and a Transformation, and recover those values from a Font:

<*fd*: Vector> <*m*: Transformation> *MAKEFONTM → <*f*: Font>
where a font is constructed from FontDescription *fd* and Transformation *m*.

<*f*: Font> *OPENFONT → <*fd*: Vector> <*m*: Transformation>
where *fd* and *m* are the FontDescription and Transformation associated with the font.

Note that the Transformation component of a Font is distinct from the *transformation* property of a FontDescription. The FontDescription's transformation converts FontDescription coordinates to character coordinates; the Font's transformation converts character coordinates to master coordinates.

The MAKEFONT operator constructs a new font from a FontDescription:

<*fd*: Vector> MAKEFONT → <*f*: Font>
where the effect is *fd* 1 SCALE *MAKEFONTM.

A typical Interpress master will refer to fonts by name, rather than constructing them. The full universal name of a font serves to identify the font unambiguously. The following operator looks up fonts in a printer's font library; it does the best it can to find a suitable font, approximating if necessary.

<*v*: Vector> FINDFONT → <*f*: Font>
where *v* is a Vector of Identifiers, which is the universal name of the desired font. The result is the best approximation to this font which the printer can find in its

environment. If the result differs from the named font, an appearance warning or appearance error occurs, depending on the closeness of the approximation.

There are numerous properties of a font that can be encoded in its name. For example:

- *Character set mapping.* The correspondence between character indices and shapes is encoded in the font name. For example, in the name *Xerox/xc2-0-0/TimesRoman*, the identifier *xc2-0-0* might be used to identify a particular mapping. If all Xerox products were to use standard mappings, then the mapping property would be associated with the part of the name *Xerox* rather than with a separate identifier.
- *Typeface.* Typeface names in the printing industry have no guaranteed structure. We find names such as "Times Roman," "Times Italic," "Helvetica Light," "Bodoni Condensed." Although it is tempting to organize these names into a rigid framework, there will always be exceptions. As a consequence, Interpress allows the font name to capture these properties in an arbitrary way.
- *Design size.* It is often desirable to use slightly different character shapes for character sizes that subtend a different angle at normal viewing distances. Characters that will be extremely small when viewed normally, such as in footnotes, often use thicker strokes than normal "body" fonts. Characters that will be unusually large, such as titles or headlines, often use narrower strokes than body fonts. These properties are quite separate from the physical size of the characters—a billboard may use "body" font characters that are 50 cm high! The font name can encode the "design size" as three discrete values (*footnote*, *body*, *headline*) or in a more continuous way (*DesignSize-9pt*). The physical size of the mask created by a character operator is determined not by its name, but rather by the transformation in effect when it is invoked.
- *Version.* Font libraries will be constantly maintained and updated. A truly unique name of a font, therefore, will include a version number, probably as the last element of the universal name. So an identifier like *version102* might be appended to the example above to indicate the version.

Fonts returned by MAKEFONT or FINDFONT contain the identity transformation; that is, they express characters in the character coordinate system. For imaging purposes, the master usually wants to transform the character coordinate system into a more convenient coordinate system. The MODIFYFONT operator allows all characters to be scaled to a particular size.

<f_1: Font> <*m*: Transformation> MODIFYFONT → <f_2: Font>
where the effect is f_1 *OPENFONT *m* CONCAT *MAKEFONTM

Usually the master saves fonts in a frame for future reference. The SETFONT operator sets the *font* imager variable from an element of the frame. SHOW (§4.9.3) can then image characters from this font.

<*n*: Cardinal> SETFONT → < >
where the effect is FGET *font* ISET; i.e., the current font is set to the *n*th element of the current frame.

The *net* transformation applied to characters in a font when they are imaged by SHOW is $nm = <m\ T$ CONCAT T_{ID}^{-1} CONCAT>, where T_{ID}^{-1} is the inverse of the ICS-to-DCS transformation (§4.3.5) and *m* is the transformation result of *OPENFONT; *nm* transforms the character coordinate system into the Interpress coordinate system. The imager may specify for each font a set of *easy* values for *nm* which it handles efficiently, and it may be unable to image a font at all unless *nm* is easy (§5.4.2).

4.9.3 Character operators

For each character in a font, there is a corresponding character operator. The following operators precisely define the effect of a character operator.

<*fd*: Vector> <*i*: Cardinal> MASKCHAR → <*fd*>
where *fd* is a FontDescription, and *i* is a character index. If a mask operator is defined for character *i*, it is executed with *fd* on the stack. The effect is:

0 MARK *fd* DUP *characterMasks* GETP *i* GETPROP { DOSAVEALL } IF
COUNT MAKEVEC POP UNMARK0 *fd*

MASKCHAR is provided so that a composite character mask can be assembled from other character masks in the same font—for example, to form an accented character. It is expected that only character mask operators in a FontDescription will explicitly call MASKCHAR. Note that the current transformation when MASKCHAR is called must transform FontDescription coordinates to device coordinates; this is the case when *DOCHAR invokes MASKCHAR.

<*f*: Font> <*i*: Cardinal> *DOCHAR → < >
where the character operator for the character with index *i* in font *f* is executed.

We say that character *i* is *in* font *f* if *f* *OPENFONT POP *characterMetrics* GETP *i* GETP succeeds. If character *i* is in *f*, the effect of *f i* *DOCHAR is:

```
{
-- Apply the character-to-master transformation; save the FontDescription. --
f *OPENFONT CONCATT 0 FSET -- FontDescription vector --
-- Apply the FontDescription-to-character transformation. --
0 FGET transformation GETP CONCATT
-- Fetch metrics for character i. --
0 FGET characterMetrics GETP i GETP 1 FSET -- CharacterMetrics vector --
1 FGET escapementX GETPROP NOT { 0 } IF 2 FSET -- escapementX, default 0 --
1 FGET escapementY GETPROP NOT { 0 } IF 3 FSET -- escapementY, default 0 --
1 FGET amplified GETPROP NOT { 0 } IF 4 FSET -- amplified, default 0 --
1 FGET correction GETPROP NOT { 0 } IF 5 FSET -- correction, default 0 --
-- Image the mask for character i. --
0 FGET i MASKCHAR POP
-- If it is an amplifying character, multiply escapements by amplifySpace. --
4 FGET {
    2 FGET amplifySpace IGET MUL 2 FSET -- escapementX*amplifySpace --
    3 FGET amplifySpace IGET MUL 3 FSET -- escapementY*amplifySpace --
    } IF
-- Adjust the current position. --
2 FGET 3 FGET SETXYREL
-- Perform spacing correction. --
5 FGET 0 EQ { CORRECTMASK } IF -- if correction=0 --
5 FGET 1 EQ { 2 FGET 3 FGET CORRECTSPACE } IF -- if correction=1 --
} DOSAVESIMPLEBODY
```

If character *i* is not in *f*, the effect of *f i* *DOCHAR is *f substitute* *DOCHAR, where *substitute* is the value of the font's *substituteIndex* property. If *substitute* is not in *f*, a master error occurs.

The SHOW operator images a string of characters in the current *font*, an imager variable, by invoking their character operators in sequence.

<*v*: Vector> SHOW → < >
where for each *i*, $l \leq i \leq u$, beginning with *v*'s lower bound *l* and ending with *v*'s upper bound *u*, the effect is

```
{ TRANS font IGET v i GET *DOCHAR } DOSAVESIMPLEBODY
```

SHOW uses the current position to establish the origins for successive character instances. Because each character operator is executed with DOSAVE, only changes to persistent variables will be visible after each step of SHOW; thus the current position does change, as it must if successive characters are to be laid down in the proper places, but any changes to the current transformation, the color, etc., are forgotten. Thus each character mask operator can use all the facilities of Interpress to make its image, without interfering with the rest of the imaging in the master.

Sometimes it is necessary to insert a positioning operation between each pair of characters, e.g., when kerning. For this purpose, the following operator is useful:

<*v*: Vector> SHOWANDXREL → < >
where alternate elements of *v* are taken as character indices and as distances to move the *x*-coordinate of the current position. SHOWANDXREL treats the first element of *v* just as SHOW would. It takes the next element modulo 256 and then biases it by 128 to yield

an argument for SETXREL. The next element is shown, and so forth. More precisely, for each i, $l \leq i \leq u$, beginning with v's lower bound l and ending with v's upper bound u, the effect is

$i\ l$ SUB 2 MOD 0 EQ { [$v\ i$ GET] SHOW } IFELSE
{ $v\ i$ GET 256 MOD 128 SUB SETXREL } IF

The reason for taking the distances modulo 256 is to discard the *offset* which might be in force if v is generated by the *sequenceString* encoding notation. The reason for the bias by 128 is to make positive and negative kerning equally convenient.

A variant of this scheme is convenient when the intercharacter spaces are all the same:

<v: Vector> <x: Number> SHOWANDFIXEDXREL → < >
where for each i, $l \leq i \leq u$, beginning with v's lower bound l and ending with v's upper bound u, the effect is
[$v\ i$ GET] SHOW $i\ u$ EQ NOT { x SETXREL } IF

4.9.4 Fallback positions for characters

The chief difficulty that an imager may encounter when printing characters is that the exact character geometry specified by the current transformation cannot be achieved. This will occur when device dependencies limit the size of characters, prevent certain rotations, etc. When this occurs, the imager should:

- Use a mask that approximates the one requested. Interpress does not specify how approximations are to be selected.
- Perform the width and correction calculations accurately, using the transformation specified. In other words, although the mask will only approximate the character shape, positioning will remain accurate.
- Generate an appearance error.

4.10 Spacing correction

Sometimes the exact positioning of a mask must be computed when the master is printed rather than when it is created. This is 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. Such width differences can arise when the imager can only approximate the font requested by the master, or if a new version of a font with slightly different widths has superseded the font in effect when the master was created. Of course, if the creator knows the properties of the imager's 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* the spacing of a set of masks, which is used most frequently to insure that lines of characters intended to appear uniformly justified between margins are in fact 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 the spacing of any kind of mask.

Note that the correction mechanism is *not* intended to be used to achieve line justification. The *amplifySpace* mechanism described in §4.9 will handle simple justification needs. More complex justification must be computed by

the creator and reflected in the master as precise character positioning. The purpose of correction is to insure that a line of text ends in the right place even when approximations have been made for the fonts used in it.

Mask correction is achieved with the CORRECT operator, which takes as its only argument a body containing the operators that invoke all the masks whose positions are to be corrected. 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 entered, the current position is noted, and the body is executed, but mask operators are not allowed to alter the page image (the variable *noImage* is set to 1). As masks are invoked, calls to CORRECTSPACE and CORRECTMASK (typically performed by *DOCHAR) record the number of opportunities for spacing correction. When execution of the body is finished, CORRECT computes the difference between the current position and the current position desired by the master. Then the current position is reset to the value noted at the beginning of the operation. The body is executed again, with mask operators allowed to change the page image, and with the CORRECTSPACE and CORRECTMASK operators instructed to change the current position incrementally so as to achieve proper mask spacing.

The discussion below presents detailed definitions of CORRECT, CORRECTMASK, and CORRECTSPACE. The overall effect of CORRECT, the interfaces to the operators, and the meanings of the imager variables *correctMX*, *correctMY*, *correctPass*, *correctShrink*, *correctTX*, and *correctTY* must be observed by an Interpress printer. However, the printer is free to use a printer-dependent algorithm for adjusting character positions to meet the required line length. Lines in the definitions that might be modified in printer-dependent ways are marked --*--.

The definitions below do, however, present one consistent method for achieving correction. The way the corrections are accomplished depends on whether the line of text must be lengthened or shortened. If it is to be lengthened, extra space will be inserted by each CORRECTSPACE operator in proportion to the size of the original (uncorrected) space. If the line must be shortened, CORRECT fits the line by shrinking the spaces (identified by CORRECTSPACE); however, a space is never allowed to shrink to less than (1 − *correctShrink*) times its former size. Any additional squeezing required is accomplished by removing space between all masks equally (CORRECTMASK).

Detailed definitions of the operators follow:

< > CORRECTMASK → < >
where the function is defined by the following informal code:

```
if correctPass = 1 then correctMaskCount := correctMaskCount + 1 --*--
else if correctPass = 2 and correctMaskCount > 0 then begin
    spx := correctMaskX/correctMaskCount; --*--
    spy := correctMaskY/correctMaskCount; --*--
    correctMaskX := correctMaskX − spx; correctMaskY := correctMaskY − spy; --*--
    correctMaskCount := correctMaskCount − 1; --*--
    DCScpx := DCScpx + spx; DCScpy := DCScpy + spy; --*--
    end
```

<*x*: Number> <*y*: Number> CORRECTSPACE → < >
where the function is defined by the following informal code:

```
-- obtain device coordinates of space --
dx, dy := Tv(x, y, T)
if correctPass = 1 then begin
  correctSumX := correctSumX + dx; correctSumY := correctSumY + dy --*-- end
else if correctPass = 2 then begin
  -- define 0/0 = 0 in the next line --
  spx: = dx*correctSpaceX/correctSumX; spy := dy*correctSpaceY/correctSumY; --*--
```

correctSumX := *correctSumX* − *dx*; *correctSumY* := *correctSumY* − *dy*; --*--
correctSpaceX := *correctSpaceX* − *spx*; *correctSpaceY* := *correctSpaceY* − *spy*; --*--
DCScpx := *DCScpx* + *spx*; *DCScpy* := *DCScpy* + *spy* --*--
end

< *b*: Body > CORRECT → < >
where the function is defined by the following informal code:
-- *save the starting position* --
correctcpx := *DCScpx*; *correctcpy* := *DCScpy*;
noImage := 1;
correctMaskCount := 0; *correctSumX* := 0; *correctSumY* := 0; --*--
correctPass := 1;
-- *Interpret all operators to compute required corrections* --
0 MARK; *b* DOSAVESIMPLEBODY; UNMARK0;
correctTargetX := *correctcpx* + *correctMX*; *correctTargetY* := *correctcpy* + *correctMY*;
-- *COMPUTECORRECTIONS *determines how to allocate space. See below.* --
*COMPUTECORRECTIONS;
DCScpx := *correctcpx*; *DCScpy* := *correctcpy*;
noImage := 0;
correctPass := 2;
-- *Interpret all operators, emit masks* --
0 MARK; *b* DOSAVESIMPLEBODY; UNMARK0;
correctPass := 0;
if *distance*(*correctTargetX*, *correctTargetY*, *DCScpx*, *DCScpy*) > *length*(*correctTX*, *correctTY*)
then error; -- CORRECT *did not properly adjust the mask positions* --
DCScpx := *correctTargetX*; *DCScpy* := *correctTargetY*;

The MARK, UNMARK pairs require that the operators in the body must leave the interpreter operand stack in the same state they find it. The bodies are called with DOSAVESIMPLEBODY, which saves all non-persistent variables. (DOSAVEALL cannot be used because the side-effect on current position, saved in a persistent variable, is required.) The body *b* should not change *noImage*.

The variables used to control spacing corrections are imager variables and some (persistent) variables shared by the CORRECT, CORRECTMASK, and CORRECTSPACE operators. These variables are summarized in Table 4.2. Note that calls on CORRECT cannot nest because only a single set of persistent variables is used to save CORRECT state.

The mask-correcting mechanism can be disabled simply by setting *correctPass* to 0. This variable is initialized to 0 when Interpress interpretation begins. Correction may be disabled in part of a sequence of masks by setting *correctPass* to 0 while generating their masks; since *correctPass* is an imager variable, it is saved and restored by DOSAVE and the like.

Note that proper operation of CORRECT depends on the operators CORRECTSPACE and CORRECTMASK being called at appropriate times. Character operators will normally make these calls. If, however, a master uses masks or spaces that are not provided by character operators, CORRECTSPACE and/or CORRECTMASK must be called explicitly. This is often the case for spaces when SETXYREL and the like are used to alter inter-character or inter-word spacing. The SPACE operator (§4.10.2) conveniently performs these calls.

The *COMPUTECORRECTIONS step mentioned above is responsible for computing the corrections that should be made to the current position during the second pass:

- The *target position* (*correctTargetX*, *correctTargetY*) is computed as the current position at the beginning of the CORRECT body plus the *measure*, as specified by *correctMX* and *correctMY*.

  This is the position where the current position *should* have ended up after the first pass. Note that *correctMX* and *correctMY* are saved in persistent variables so that an operator inside the body *b* can set them. This allows

a creator that generates a master in a single sequential stream to specify the target after creating the mask operators that comprise the body, as illustrated in the following example:

```
{ 3/2 amplifySpace ISET
  <This is a string.> SHOW
  133 0 SETCORRECTMEASURE
} CORRECT
```

- If the current position is short of the target position, the mask adjustments are set to zero, and the space adjustments are set so that all adjustments will sum to the difference between the target and the current position. In this way, during pass 2 the current position will end up at the target.
- If the current position lies beyond the target position, the line is compressed by first adjusting spaces until the ratio of the space adjustment to the available space exceeds the imager variable *correctShrink*, and then adjusting masks to achieve the proper length.

This calculation is stated more precisely below. First, we define some functions that measure distances on the page:

distance(x_1, y_1, x_2, y_2) = *the distance in meters between device coordinates* (x_1, y_1) *and* (x_2, y_2).
length(*dx*, *dy*) = *distance*(0, 0, *dx*, *dy*).

The *COMPUTECORRECTIONS calculation itself is:

```
correctMaskX := 0; correctMaskY := 0; --*--
correctMaskCount := correctMaskCount-1; --*--
correctSpaceX := correctTargetX-DCScpx; --*--
correctSpaceY := correctTargetY-DCScpy; --*--
-- Test if line too long and space-correction threshold is exceeded --
if length(correctSpaceX, correctSpaceY) > --*--
    correctShrink*length(correctSumX, correctSumY) and --*--
distance(correctcpx, correctcpy, correctTargetX, correctTargetY) < --*--
    distance(correctcpx, correctcpy, DCScpx, DCScpy) then begin --*--
    -- Must reposition masks too --
    correctMaskX := correctSpaceX+correctShrink*correctSumX; --*--
    correctMaskY := correctSpaceY+correctShrink*correctSumY; --*--
    correctSpaceX := correctSpaceX-correctMaskX; --*--
    correctSpaceY := correctSpaceY-correctMaskY; --*--
    if correctSumX=0 and correctSpaceX≠0 then --*--
        begin correctMaskX := correctSpaceX; correctSpaceX := 0 end; --*--
    if correctSumY=0 and correctSpaceY≠0 then --*--
        begin correctMaskY := correctSpaceY; correctSpaceY := 0 end; --*--
    end
```

The reason for subtracting 1 from *correctMaskCount* is that there are only $n-1$ opportunities to correct the spacing between n masks. Note that the *COMPUTECORRECTIONS calculations handle x and y symmetrically. They will correct lines running at any angle.

4.10.1 Efficiency

Creators are urged to use the CORRECT operator freely in order to minimize distortions caused by printing a master on a printer that cannot match exactly the font escapements assumed by the creator. However, in the cases where creator and printer are in exact agreement about escapements, the two-pass CORRECT operator seems wasteful. Interpress provides a mechanism to avoid two passes in those cases where the target position is achieved within a distance tolerance. After the first pass, if the distance between the target and the current position is less than a tolerance, i.e.,

$$distance(correctTargetX, correctTargetY, DCScpx, DCScpy) \leq length(correctTX, correctTY)$$

then the second pass need not be undertaken. Of course, if the second pass is omitted, the imager must in fact emit the masks specified, i.e., must act as if *noImage* had been false during the first pass. The tolerance is set by the imager variables *correctTX* and *correctTY*, which are initialized to printer-dependent values.

An Interpress program showing the one-pass version of CORRECT cannot be stated precisely using only imager primitives. However, the behavior of this program must match that given for CORRECT, above. The idea behind the one-pass algorithm is that the imager would save, during the first pass, a list of all page image modifications specified, but would not actually make the modifications. If, at the end of the first pass, the correction required is less than the threshold, the image modifications saved in the list can be made safely.

4.10.2 Operators

Two operators set the coordinate parameters of the CORRECT operator. Note that they are transformed as "vectors":

<*x*: Number> <*y*: Number> SETCORRECTMEASURE → < >
where $(correctMX, correctMY) := T_v(x, y, T)$.

<*x*: Number> <*y*: Number> SETCORRECTTOLERANCE → < >
where $(correctTX, correctTY) := T_v(x, y, T)$.

The following operator should be used instead of SETXREL when the creator explicitly computes the width of the spaces needed to justify a line, instead of using the *amplifySpace* mechanism.

<*x*: Number> SPACE → < >
where the effect is DUP SETXREL 0 CORRECTSPACE; i.e., the current position is changed by SETXREL and the proper call to CORRECTSPACE is made.

Table 4.2 **Variables used by correction operators**

Name	Type	Use
Imager variables (persistent):		
correctMX, *correctMY*	Number	Line measure
Imager variables (non-persistent):		
correctPass	Cardinal	
correctShrink	Number	Allowable space shrink
correctTX, *correctTY*	Number	Line tolerance
CORRECT **variables (persistent)**, not directly available to the master:		
correctMaskCount	Cardinal	Tally CORRECTMASK calls
correctMaskX, *correctMaskY*	Number	Space to be taken up by CORRECTMASK calls
correctSumX, *correctSumY*	Number	Tally adjustable space from CORRECTSPACE calls
correctSpaceX, *correctSpaceY*	Number	Space to be taken up by CORRECTSPACE calls
correctcpx, *correctcpy*	Number	Current position at start of CORRECT
correctTargetX, *correctTargetY*	Number	Where corrected text should end up

5. PRAGMATICS

This chapter deals with various practical issues connected with the implementation and operation of creators and imagers.

5.1 Printer capabilities

An Interpress master describes precisely the ideal appearance of a document in a device-independent manner. Each image output device renders its best approximation to the ideal represented by the master. Actual renditions of a document on different output devices might look different due to the capabilities of the output devices themselves.

For example, a bi-level raster device such as a laser printer must adjust the positions of characters and other graphic objects to align with its raster grid, and must use a spatially dispersed pattern of black and white pixels to produce the effect of a constant gray color.

Interpress provides extensive facilities for describing images such that many printers are not able to produce all the images which can be specified. There are three factors which determine what images a printer can produce:

- The *set* of types and primitives that it supports (§5.2).
- The operators, fonts, and colors that can be obtained from its *environment* (§5.3).
- The *complexity* of the images it can handle (§5.4).

A printer's document handling and finishing capabilities (§5.5) also constrain what documents it can produce. It is beyond the scope of this standard to specify how each printer should process every Interpress master in the face of these constraints, but the objective should be to render the best possible approximation to the effect defined by the master for as much of the document content as possible, even if this entails severe appearance errors on parts of the document content which are beyond the capability of the printer.

If a printer does not support all of the types and primitives of the language, it should nonetheless attempt to process any master it receives by stepping past those operations which it does not support. The ideal is for any Interpress printer to be able to process any master, even though the resulting output may not contain everything that was described in the master.

5.2 Interpress sets

One measure of a printer's imaging capability is the set of types and primitives that it supports together with any restrictions it places on the state of the machine when these primitives are executed. There are three standard Interpress *sets* to meet the requirements of different printing applications which the printer may serve.

- The *commercial set* supports applications requiring text, forms, and scanned images. It includes text at 90° rotations, horizontal and vertical lines, filled rectangles, and binary pixel arrays.

- The *publication set* includes all of the facilities of the commercial set, plus synthetic graphics and rectangular clipping. It includes trajectories composed of straight and/or curved lines, solid and dashed strokes, filled outlines,rectangular clipping, and solid gray scale color.

- The *professional graphics set* contains all of the imaging facilities in the language. To the facilities of the publication set it adds arbitrary rotations of text and graphics, gray scale pixels (for process color) and full color (with the appropriate environment), and arbitrary clipping.

Table 5.1 provides a general definition of these sets: they are defined precisely in the sections which follow. Printers which do not support the entire language are not restricted to one of the standard sets, but their functionality is characterized to creators by the set they support. Further capabilities, such as the types of image decompression and color which the printer can support, depend on the printer's environment (§5.3).

Table 5.1 **Interpress sets**

	Commercial Set	*Publication Set*	*Professional Graphics Set*
Base Language	all facilities	all facilities	all facilities
Text	all facilities at 90° rotations	all facilities at 90° rotations	all facilities at all rotations
Graphics	horizontal and vertical filled rectangles no clipping	strokes and curves dashed lines filled outlines rectangular clipping	strokes and curves dashed lines filled outlines arbitrary clipping
Pixel Arrays	binary pixel arrays	binary pixel arrays	gray scale pixel arrays
Color	solid and sampled black	gray scale and constant color	all colors and color operators

Some operators in set *S* are defined in this document in terms of more general operators not available in *S*, usually because they represent special cases of the more general operators. For example, MASKRECTANGLE, in the commercial set, is defined in terms of MASKFILL, which is not in that set.

5.2.1 Commercial set

The *commercial set* consists of the following facilities:

- All the types, literals, and operators of the base language (§2.2-4), and all of the facilities of the Xerox encoding (§2.5).

  The use of sequence types which specify decompression operators is subject to the constraints of the printer's environment (§5.3).

- Skeleton processing for single-level skeletons (i.e., content ::= body) (§3.1) and printing instructions (§3.3 and Appendix E).

  Support for document handling and finishing instructions is subject to the printer's capabilities (§5.5).

- All of the facilities for imager variables, transformations whose rotation components are multiples of 90°, and current position (§4.2-5).

  The restrictions below allow certain imager variables to be ignored: restrictions on *color* ensure that *priorityImportant* is always unimportant; restrictions on MASK operators ensure that *strokeJoint* is unimportant; and restrictions on clipping ensure that *clipper* must always be the full field.

- Sample decompression and the construction of binary pixel arrays by MAKEPIXELARRAY with $samplesPerPixel = 1$, $maxSampleValue = 1$, and $samplesInterleaved = 1$ (§4.6).

  The use of FINDDECOMPRESSOR is subject to the constraints of the printer's environment (§5.3).

- Solid black color and sampled colors generated by MAKESAMPLEDBLACK with $clear = 1$ (§4.7). Only the mask operators listed below can be called when a sampled color is in effect. FINDCOLOR subject to the constraint that the "best approximation" to any constant color may be black.

  These restrictions ensure that priority is unimportant and that the only output transition function required for a bilevel raster printer is to "OR" the pixels from a mask operation with the image pixels.

- Horizontal and vertical rectangles masked by the operators MASKRECTANGLE, STARTUNDERLINE, MASKUNDERLINE, and MASKVECTOR with $strokeEnd = 0$ or $strokeEnd = 1$ and either $x_1 = x_2$ or $y_1 = y_2$ (§4.8).

  The Outline and Trajectory types and the operators which generate them are not included.

- Pixel arrays imaged via MASKPIXEL with *color*= <1 MAKEGRAY> (black) (§4.8).

- All of the text imaging operations of §4.9-10 (characters and correction) with the exception of MAKEFONT and MASKCHAR.

5.2.2 Publication set

The *publication set* consists of all of the facilities of the commercial set plus facilities for trajectories composed of straight and/or curved lines, solid and dashed strokes, filled outlines, and rectangular clipping.

- All the types, literals, and operators of the base language (§2.2-4), and all of the facilities of the Xerox encoding (§2.5).

  The use of sequence types which specify decompression operators is subject to the constraints of the printer's environment (§5.3).

- All the facilities for skeleton processing (§3.1) and printing instructions (§3.1 and Appendix E).

Support for document handling and finishing instructions is subject to the printer's capabilities (§5.5).

- All of the facilities for imager variables, transformations whose rotation components are multiples of 90°, and current position (§4.2-5).

  The restrictions on transformations and clipping operators ensure that *clipper* is either the full field or a rectangle whose edges are parallel to the axes of the Interpress coordinate system.

- Sample decompression and the construction of binary pixel arrays by MAKEPIXELARRAY with *samplesPerPixel*=1, *maxSampleValue*=1, and *samplesInterleaved*=1 (§4.6).

  The use of FINDDECOMPRESSOR is subject to the constraints of the printer's environment (§5.3).

- The color operators MAKEGRAY, FINDCOLOR, MAKESAMPLEDBLACK, SETGRAY, and SETSAMPLEDBLACK (§4.7). All mask operators except MASKPIXEL can be called with any color in effect.

  Constant colors which may be found by FINDCOLOR are subject to the constraints of the environment. The FINDCOLOROPERATOR, FINDCOLORMODELOPERATOR, MAKESAMPLEDCOLOR, and SETSAMPLEDCOLOR operators are not included.

- Trajectories composed of straight and/or curved line segments; solid and dashed strokes with all joint and end types; filled outlines; and CLIPRECTANGLE (§4.8).

- Pixel arrays imaged via MASKPIXEL with a constant color in effect (§4.8).

- All of the text imaging operations of §4.9-10 (characters and correction) with the exception of MAKEFONT and MASKCHAR.

5.2.3 Professional graphics set

The *professional graphics set* consists of all of the imaging facilities of the language.

- All the types, literals, and operators of the base language (§2.2-4), and all of the facilities of the Xerox encoding (§2.5).

  The use of sequence types which specify decompression operators is subject to the constraints of the printer's environment (§5.3).

- All the facilities for skeleton processing (§3.1) and printing instructions (§3.3 and Appendix E).

  Support for document handling and finishing instructions is subject to the printer's capabilities (§5.5).

- All of the facilities of imager variables, transformations, and current position (§4.2-5).

- All facilities for sample decompression and pixel array construction (§4.6). Sample vectors may have multiple samples per pixel, scaled (halftone) sample values, and samples interleaved or not.

- All constant colors and the ability to make sampled colors using gray scale or full color pixels. Sampled colors may be used with any mask operator or for printing text or graphics (§4.7).

  Constant colors which may be found by FINDCOLOR and color operators found by FINDCOLOROPERATOR and FINDCOLORMODELOPERATOR are subject to the constraints of the environment. Some printers may support only gray scale colors and possibly highlight colors.

- Trajectories composed of straight and/or curved line segments; solid and dashed strokes with all joint and end types; filled outlines with any *color*; and arbitrary clipping (§4.8).
- Pixel arrays imaged via MASKPIXEL with any *color* in effect (§4.8).
- All of the text imaging operations of §4.9-10 (characters and correction), including the ability to build and image fonts at run time using MAKEFONT and MASKCHAR.

5.3 Environment

Interpress imposes no requirements on a printer's environment. Each printer must determine what its environment contains.

Other standards may specify these requirements. For example, the Xerox printing environment is specified by the Xerox Print Service Integration Standard. This standard also specifies font naming conventions, the file name encoding and syntax used for *sequenceInsertMaster* and *sequenceInsertFile* tokens, and the protocols used for transmitting and printing documents.

References to the environment are of two types: by the execution of the primitives FINDOPERATOR, FINDFONT, FINDCOLOR, FINDCOLOROPERATOR, FINDCOLORMODELOPERATOR, or FINDDECOMPRESSOR, and by means of the *sequenceInsertMaster* or *sequenceInsertFile* encoding-notations. The result of an attempt to FIND an object which cannot be obtained from the printer's environment depends on the effect on the Interpress interpreter's ability to produce the desired page image and to continue to correctly interpret the master. The execution of a FINDFONT or a FINDCOLOR primitive is guaranteed to be successful in the sense that the printer is required to provide an approximation to the named object if it cannot obtain the object itself. The effect of a FINDFONT or a FINDCOLOR primitive which names an object which cannot be obtained is therefore either an appearance error or an appearance warning, depending on the closeness of the approximation.

The result of an unsuccessful attempt to find an operator with a FINDOPERATOR. FINDCOLOROPERATOR, FINDCOLORMODELOPERATOR, or FINDDECOMPRESSOR primitive is a master error. Mark recovery occurs unless the Interpress interpreter is able to continue processing the master despite its inability to obtain the missing operator.

The result of a *sequenceInsertMaster* or a *sequenceInsertFile* token which names a file which cannot be obtained is a master error. Since the effect of a *sequenceInsertMaster* token is to insert complete pages, mark recovery does not occur. A *sequenceInsertFile* which names a file which cannot be obtained results in mark recovery.

5.4 Complexity

This section addresses issues related to the complexity of the page images which a printer can render.

- The number of characters or other graphic objects that can be rendered on a page, and the limitations on local complexity (§5.4.1).

- The pixel arrays which can be converted to sampled colors or use as masks (§5.4.2).
- The effect of image complexity on printer performance (§5.4.3).

5.4.1 Image complexity

The Interpress standard does not limit the complexity of the images to be printed. From a user perspective it is desirable to have printers which can print images of "unlimited" complexity, even if performance is greatly degraded on complex images. This is because even if most pages of a master are of moderate complexity, only one very complex page is needed to render the entire master unprintable. But there is actually no such thing as truly "unlimited" printing: practical limits of memory size or processor speed impose limitations on the complexity of the images which a particular printer can print, and for every printer there is some page image which is too complex to be properly rendered. What can be accomplished is to build printers whose limits are so high that they are able to render "practically" all of the images that they are asked to print.

Many physical printing devices operate synchronously for some unit of output, called a *block*; that is, once output of a block has started, it must continue at a fixed or minimum rate. An asynchronous device has no such requirement, and can normally be used to print images of unlimited complexity. To make a printer with a synchronous device which can print images of unlimited complexity, however, the printer must contain enough buffering to construct and store all the output for a block. For example, xerographic devices typically have one-page blocks, and the output to the device is in raster form; when such a device is used in an unlimited printer, a buffer which can store all the bits in a one-page raster is required. At 300 pixels/in. about 13×10^6 bits of buffering are required for a 9×14 inch page.

To use a printer which is relatively limited in the images it can handle, one must have some useful characterization of its maximum image complexity. Unfortunately, this characterization usually involves complex local properties of the image which are not easy to state even for a particular printer, much less in general.

5.4.2 Easy net transformations

A printer may specify the net transformations of pixel arrays (§4.6) or fonts (§4.9) from their standard coordinate systems to the Interpress coordinate system that the printer can handle efficiently; these are the *easy* net transformations, which typically contain only scaling and rotation components. While a printer is assumed to be able to handle arbitrary translations, it may refuse to handle a sampled color, pixel array mask, or font operator which has a net transformation not in the easy set.

Interpress does not define how easy transformations are specified. For fonts, the Xerox Font Interchange Standard describes how to specify easy transformations in a FontDescription.

5.4.3 Performance

Not all masters are equally easy to execute because different operators and contructs entail different amounts of computing. Moreover, certain printers may process certain masters efficiently, while slowing down for others. Interpress does not specify performance requirements: documentation for a printer may specify its performance and the properties that a master should have to be executed efficiently. A printer must support all facilities defined for its set, but is free to support only some of these facilities efficiently.

5.5 Document handling and finishing

Interpress provides a means of specifying a variety of document handling and finishing options which many printers will be unable to provide. The following comments deal with document handling and finishing capabilities.

- An Interpress master describes a document as consisting of (possibly a large number of) copies which may differ in the pages which are part of each copy (*pageSelect*, §3.1) and in minor ways in the content of certain of these pages (IFCOPY, §2.4.7). Limitations on internal storage may restrict the size of documents for which multiple copies may be printed in collated sets.

  If a multiple copy job is too large to be spooled in its entirety, the printer may print it as collated page sets or in uncollated form. Some printers will be unable to provide collation on *any* multiple copy jobs, and may even be unable to print multiple copy jobs at all. Selection of copy number by *copySelect*, selection of pages by *contentSelect*, and *pageSelect*, and selection of page content by IFCOPY would still be supported.

- A printer which can only print simplex would impose *plex = simplex* as a printer override in *ADDINSTRUCTIONDEFAULTS.

  Values for *onSimplex* and *pageOnSimplex* would still be supported.

- The number of different media which may be used to print a single document depends on the printer's media handling facilities.

  A printer might reasonably accept only as many values of *mediumDescription* in the *media* property vector as it could reasonably handle. Additional values would then be ignored and values of *mediaSelect*, *contentMediaSelect*, and *pageMediaSelect* which refer to these media mapped to 1.

- A printer which does not have separate output bins or an offset stacker will ignore the *outputPosition* and *contentOutputPosition* instructions.

  Even a printer which does have these facilities will be limited in the number of distinct output positions which it can provide.

- Support for the various values of *finishing* depends in general on the availability of the appropriate finishing hardware. A printer which does not have the requisite capabilities would ignore those values.

5.6 Numeric precision and size limits†

Numbers in Interpress are used mainly for computing device coordinates from master coordinates and transformations (§4.3). This section gives the rules which text Interpress masters must observe to ensure that device coordinates are computed accurately enough to produce good images. If the rules are violated, it is possible that bad images will be produced because the wrong device coordinate values may be used. In most cases, however, there will be no detected error, and it is possible that the image will still be acceptable. If the rules are observed, however, it is guaranteed that device coordinates will be computed accurately, in the sense defined below. The limits on the size of device coordinates and Numbers do not have this fail-soft property, since the numbers may overflow the representation if they are too big.

Limits on Numbers:

The absolute value of a Number must not be larger than 10^{20}.

Limits on the size of device coordinates:

When the current transformation T is applied to a pair of coordinates (c_x, c_y) to produce a pair of absolute device coordinates (d_x, d_y), the magnitude of the d's must be less than the largest field dimension (§4.3.1) plus 10%. Furthermore, the magnitude of the relative device coordinates produced by $T_v(c_x, c_y, T)$ must be within this range.

Limits on sequences of relative moves:

The total path length, in device coordinates, of a sequence of relative moves must be within this range. A sequence of relative moves must be limited to 250 moves.

Limits on transformations:

Let T' be the upper left 2×2 part of a transformation T, s_{min} the singular value of T' with smallest magnitude, and s_{max} the other one (the singular values of a matrix M are the positive square roots of the eigenvalues of MM^T). Then $s_{max}/s_{min}<16$, $s_{max}<10^{20}$, and $s_{min}>10^{-20}$. A transformation must not be computed by concatenating more than 8 primitive transformations obtained from TRANSLATE, ROTATE, SCALE, and SCALE2. The product of s_{max}/s_{min} for the primitive transformations concatenated together must be less than 16.

The restrictions on numbers and device coordinates prevent overflow. The restrictions on relative moves and transformations allow the loss of precision in computing device coordinates to be bounded. If these restrictions are observed, the imager guarantees that a computed device coordinate will not differ from its ideal value by more than 1/4 of a grid unit (§4.3.4). Furthermore, the Number value resulting from a literal will be as close to the rational number represented by the literal as the nearest IEEE floating point number, and each primitive operation on Numbers will produce a result which is as close to the ideal result as the nearest IEEE floating point number.

5.6.1 Size limits

Printers are also limited in the size of the objects and values they can manipulate. The following table gives minimum values for the size limits defined in this standard.

Table 5.2 **Minimum values for size limits**

Name	Where defined	Minimum limit
maxCardinal	(§2.2.1)	2^{24}-1
maxIdLength	(§2.2.2)	100 characters
maxBodyLength	(§2.2.5)	10000 literal
maxStackLength	(§2.3.1)	1000 values
maxVecSize	(§2.2.4)	1000 elements
maxFileNesting	(§2.5.3)	8 files
topFrameSize	(§3.1)	50 elements

5.7 Error handling

The effect of a master error on the execution of an Interpress master is defined in §2.4.1. In addition, however, a printer should provide some indication of what errors have occurred, and how severe they are. This section offers guidance in this matter.

As an Interpress master is printed, various errors may be encountered. The first few errors should be reported by printing an explanatory message. This message should indicate at least:

- the page number;
- the current position;
- the composed operator being executed if it is not a page image body;
- the severity of the error, as defined below;
- some indication of the nature of the error.

If there is not enough space to report all the errors, this fact should be reported. If possible, each page on which an error occurs should be identified.

Errors are classified according to their severity:

Appearance warning:

These errors mean that the imager had to make an approximation to the ideal image represented in the Interpress master, but has been able to preserve the content of the image.

Examples are the substitution of a font which is a close approximation to that called for in the master, or reduction in size of the entire image compared to the size specified.

Appearance error:

These errors mean that 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.

An example is failure of the imager to display a filled mask or a pixel array that is represented in the master.

Master warning:

These errors mean that something is amiss in the specification of the master, but the error is not severe and the interpreter is able to continue processing the master properly.

For example, arithmetic overflow will cause a master warning. Another example would be the attempt to use a decompressor which cannot be obtained, provided that the interpreter is able to provide a "substitute" decompressed pixel array and to continue to interpret the master.

Master error:

These errors signal severe problems in interpreting the master. In case of a master error in executing an operator, unless the operator definition specifies otherwise, there is a mark recovery (§2.4.1). In some cases, it may be necessary to abandon further interpretation of the master and simply to print a page that describes the error.

A. REFERENCES

A Proposed Standard for Binary Floating-Point Arithmetic. *Computer*, **14**, 3, March 1981, p 51.

A draft of the reference document for the proposed IEEE standard. The same issue of *Computer* also contains other articles about the standard.

Coonen, J.T. An implementation guide to a proposed standard for floating-point arithmetic. *Computer*, **13**, 1, January 1980, p 68.

A discussion of the proposed IEEE floating-point standard. A list of errata for this article appears in the preceding reference.

International Standards Organization. *7-Bit Coded Character Set for Information Processing Interchange*. ISO 646–1983 (E).

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

Newman, W.M. and Sproull, R.F., *Principles of Interactive Computer Graphics*, 2nd edition, McGraw-Hill, 1979.

Introduction to computer graphics, geometric representations and transformations, and raster graphics.

Xerox Corporation. *Character Code Standard*. Xerox System Integration Standard. Stamford, Connecticut; 1984 April; XNSS 058404 (XSIS 058404).

An enumeration of characters and their numeric codes. Also describes a mechanism for a compact encoding of strings of character codes.

Xerox Corporation. *Interpress 82 Reader's Guide*. Xerox System Integration Guide. Stamford, Connecticut; 1984 April; XNSG 018404 (XSIG 018404).

An overview of Interpress, including a paragraph-by-paragraph commentary on portions of the standard. XNSG 018404 corresponds to an earlier version of the standard; most of its contents, however, apply to Interpress 3.0 as well.

Xerox Corporation. *Introduction to Interpress*. Xerox System Integration Guide. Stamford, Connecticut; 1984 April; XNSG 038404 (XSIG 038404).

Comprehensive tutorial on Interpress, intended for system designers and programmers writing creator software.

Xerox Corporation. *Raster Encoding Standard*. Xerox System Integration Standard. Stamford, Connecticut; 1985 June; XNSS 178506.

Describes a standard form of encoding for sampled images, plus several Xerox decompression operators and color model operators.

Xerox Corporation. *Print Service Integration Standard*. Xerox System Integration Standard. Stamford, Connecticut; version 2.0, in process.

Defines the Xerox printing environment, including file name encoding and syntax, font naming syntax, and standard objects in the environment.

B.1 Types

The following types are defined in Interpress. A – indicates that the type has no code.

Name	TYPE code	Section	Comments
Any	–	2.2	any type except Body or Mark
Body	–	2.2.5	can only be the immediate argument of a body operator
Cardinal	–	2.2.1	Subtype of Number
Clipper	11	4.8	
Color	7	4.7	
ConstantColor	–	4.7.1	subtype of Color
Font	10	4.9	
Identifier	2	2.2.2	
Mark	–	2.2.3	can only be the argument of UNMARK or COUNT
Number	1	2.2.1	
Operator	4	2.2.5	
Outline	9	4.8	
PixelArray	6	4.6	
Trajectory	8	4.8	
Transformation	5	4.4	
Vector	3	2.2.4	

B.2 Primitive operators, ordered by function

A prefixed * means that the operator is *special*; it cannot be called from an Interpress master.

Vectors (§2.4.3):

GET MAKEVECLU MAKEVEC SHAPE GETPROP GETP MERGEPROP

Frames (§2.4.4):

FGET FSET

Operators (§2.4.5):

MAKESIMPLECO DO DOSAVE DOSAVEALL DOSAVESIMPLEBODY FINDOPERATOR

Stack (§2.4.6):

POP COPY DUP ROLL EXCH MARK UNMARK UNMARK0 COUNT NOP ERROR

Control (§2.4.7)

IF IFELSE *COPYNUMBERANDNAME IFCOPY

Test (§2.4.8):

EQ *EQN GT GE AND OR NOT TYPE

Arithmetic (§2.4.9):

ADD SUB NEG ABS FLOOR CEILING TRUNC ROUND MUL DIV MOD REM

Skeleton (§3.1)

*PGET *PSET *ISBODY *MAKECOWITHFRAME *LASTFRAME *OBTAINEXTERNALINSTRUCTIONS *ADDINSTRUCTIONDEFAULTS *RUNSIZE *RUNGET *MERGECONTENTINSTRUCTIONS

Imager state (§4.2):

IGET ISET *SETMEDIUM

Coordinate systems (§4.3):

*DROUND

Transformations (§4.4)

MAKET TRANSLATE ROTATE SCALE SCALE2 CONCAT CONCATT MOVE TRANS

Current position (§4.5):

SETXY SETXYREL SETXREL SETYREL GETCP

Pixel arrays (§4.6):

MAKEPIXELARRAY EXTRACTPIXELARRAY FINDDECOMPRESSOR

Color (§4.7):

MAKEGRAY FINDCOLOR FINDCOLOROPERATOR FINDCOLORMODELOPERATOR MAKESAMPLEDCOLOR MAKESAMPLEDBLACK SETGRAY SETSAMPLEDCOLOR SETSAMPLEDBLACK

Masks (§4.8):

MOVETO LINETO LINETOX LINETOY CURVETO CONICTO ARCTO MAKEOUTLINE MAKEOUTLINEODD MASKFILL MASKRECTANGLE STARTUNDERLINE MASKUNDERLINE MASKTRAPEZOIDX MASKTRAPEZOIDY MASKSTROKE MASKSTROKECLOSED MASKVECTOR MASKDASHEDSTROKE MASKPIXEL CLIPOUTLINE CLIPRECTANGLE

Characters (§4.9):

*MAKEFONTM *OPENFONT MAKEFONT FINDFONT MODIFYFONT SETFONT MASKCHAR *DOCHAR SHOW SHOWANDXREL SHOWANDFIXEDXREL

Corrected masks (§4.10):

CORRECTMASK CORRECTSPACE CORRECT *COMPUTECORRECTIONS SETCORRECTMEASURE SETCORRECTTOLERANCE SPACE

B.3 Primitive operators, ordered alphabetically

The last five columns of the table summarize useful information about each operator:

SECTION: the section in which the operator is defined.

ENCODING VALUE: the decimal integer value used to represent the operator in the encoding.

VARIABLE STACK: the operator takes a variable number of arguments or returns a variable number of results.

BODY OPERATOR: the operator takes a body as its last argument.

REDUNDANT: the operator is an abbreviation for a *simple* Interpress program.

OPERATOR	SECTION	ENCODING VALUE	VARIABLE STACK	BODY OPERATOR	REDUN-DANT
ABS	2.4.9	200			•
ADD	2.4.9	201			
*ADDINSTRUCTIONDEFAULTS	3.1	–			
AND	2.4.8	202			•
ARCTO	4.8.1	403			•
CEILING	2.4.9	203			•
CLIPOUTLINE	4.8.5	418			
CLIPRECTANGLE	4.8.5	419			•
*COMPUTECORRECTIONS	4.10	–			
CONCAT	4.4.3	165			
CONCATT	4.4.5	168			•
CONICTO	4.8.1	404			
COPY	2.4.6	183	•		
*COPYNUMBERANDNAME	2.4.7	–			
CORRECT	4.10	110		•	
CORRECTMASK	4.10	156			
CORRECTSPACE	4.10	157			
COUNT	2.4.6	188	•		
CURVETO	4.8.1	402			
DIV	2.4.9	204			
DO	2.4.5	231	•		
*DOCHAR	4.9.3	–			
DOSAVE	2.4.5	232	•		
DOSAVEALL	2.4.5	233	•		
DOSAVESIMPLEBODY	2.4.5	120	•	•	•
*DROUND	4.3.5	–			
DUP	2.4.6	181			•
EQ	2.4.8	205			
*EQN	2.4.8	–			
ERROR	2.4.6	600			
EXCH	2.4.6	185			•
EXTRACTPIXELARRAY	4.6	451			
FGET	2.4.4	20			
FINDCOLOR	4.7.1	423			
FINDCOLORMODELOPERATOR	4.7.1	422			

OPERATOR	SECTION	ENCODING VALUE	VARIABLE STACK	BODY OPERATOR	REDUN-DANT
FINDCOLOROPERATOR	4.7.1	421			
FINDDECOMPRESSOR	4.6.1	149			
FINDFONT	4.9.1	147			
FINDOPERATOR	2.4.5	116			●
FLOOR	2.4.9	206			●
FSET	2.4.4	21			
GE	2.4.8	207			●
GET	2.4.3	17			
GETCP	4.5	159			
GETP	2.4.3	286			●
GETPROP	2.4.3	287			●
GT	2.4.8	208			
IF	2.4.7	239	●	●	
IFCOPY	2.4.7	240		●	
IFELSE	2.4.7	241	●	●	●
IGET	4.2	18			
*ISBODY	3.1	–			
ISET	4.2	19			
*LASTFRAME	3.1	–			
LINETO	4.8.1	23			
LINETOX	4.8.1	14			●
LINETOY	4.8.1	15			●
*MAKECOWITHFRAME	3.1	–			
MAKEFONT	4.9.2	150			
*MAKEFONTM	4.9.2	–			
MAKEGRAY	4.7.1	425			
MAKEOUTLINE	4.8.1	417	●		
MAKEOUTLINEODD	4.8.1	416	●		
MAKEPIXELARRAY	4.6	450			
MAKESAMPLEDBLACK	4.7.2	426			●
MAKESAMPLEDCOLOR	4.7.2	427			
MAKESIMPLECO	2.4.5	114		●	
MAKET	4.4.3	160			
MAKEVEC	2.4.3	283	●		
MAKEVECLU	2.4.3	282	●		
MARK	2.4.6	186	●		
MASKCHAR	4.9.3	140			
MASKDASHEDSTROKE	4.8.3	442			
MASKFILL	4.8.2	409			
MASKPIXEL	4.8.4	452			
MASKRECTANGLE	4.8.2	410			●
MASKSTROKE	4.8.3	24			
MASKSTROKECLOSED	4.8.3	440			
MASKTRAPEZOIDX	4.8.2	411			●
MASKTRAPEZOIDY	4.8.2	412			●
MASKUNDERLINE	4.8.2	414			●
MASKVECTOR	4.8.3	441			●
MERGECONTENTINSTRUCTIONS	3.1	–			
MERGEPROP	2.4.3	288			

OPERATOR	SECTION	ENCODING VALUE	VARIABLE STACK	BODY OPERATOR	REDUN-DANT
MOD	2.4.9	209			•
MODIFYFONT	4.9.2	148			
MOVE	4.4.5	169			•
MOVETO	4.8.1	25			
MUL	2.4.9	210			
NEG	2.4.9	211			•
NOP	2.4.6	1			•
NOT	2.4.8	212			•
*OBTAINEXTERNALINSTRUCTIONS	3.1	–			
*OPENFONT	4.9.2	–			
OR	2.4.8	213			•
*PGET	3.1	–			
POP	2.4.6	180			
*PSET	3.1	–			
REM	2.4.9	216			•
ROLL	2.4.6	184	•		
ROTATE	4.4.3	163			
ROUND	2.4.9	217			•
*RUNGET	3.3.2	–			
*RUNSIZE	3.3.2	–			
SCALE	4.4.3	164			•
SCALE2	4.4.3	166			•
SETCORRECTMEASURE	4.10.2	154			•
SETCORRECTTOLERANCE	4.10.2	155			•
SETFONT	4.9.2	151			•
SETGRAY	4.7.3	424			•
*SETMEDIUM	4.2	–			
SETSAMPLEDBLACK	4.7.3	428			•
SETSAMPLEDCOLOR	4.7.3	429			•
SETXREL	4.5	12			•
SETXY	4.5	10			•
SETXYREL	4.5	11			•
SETYREL	4.5	13			•
SHAPE	2.4.3	285			
SHOW	4.9.3	22			
SHOWANDFIXEDXREL	4.9.3	145			•
SHOWANDXREL	4.9.3	146			•
SPACE	4.10.2	16			•
STARTUNDERLINE	4.8.2	413			•
SUB	2.4.9	214			
TRANS	4.4.5	170			•
TRANSLATE	4.4.3	162			•
TRUNC	2.4.9	215			
TYPE	2.4.8	220			
UNMARK	2.4.6	187	•		
UNMARK0	2.4.6	192			•

C. INTERPRESS NAME REGISTRY

Organizations wishing to construct universal names that can be referenced reliably from any Interpress master should apply for a universal identifier in the Interpress Universal Registry by contacting:

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

There is also a registry for universal names whose first identifier is the universal identifier *standard*. Individuals and organizations wishing to have "standard" universal names assigned should contact the Interpress Registry at the above address.

D. CHANGE HISTORY

This appendix presents a brief description of the principal changes that have been made in the Interpress standard.

The following changes convert Interpress version 1.0 to version 2.0:

§2.4.1. Error recovery is simplified and mark recovery defined more precisely.

§2.4.3. The operators GET, SHAPE, GETPROP, and MERGEPROP are added; the operator *GET is superseded by GET. The definitions of property vectors and universal property vectors are added.

§2.4.8. The operator *EQN is added.

§2.5. The version number in the header is changed from 1.0 to 2.0.

§2.5.3. The specification of the encoding of *sequencePackedPixelVector* and *sequenceCompressedPixelVector* is changed slightly.

§3. Printing instructions are added, resulting in a new section §3.3 and numerous small changes elsewhere in §3.

§4.2. The type of the value passed to ISET must match the type of the corresponding imager variable. The type of *correctPass* is changed to Integer.

§4.3.1. An interpretation is given to printing on the back side of a page when printing on both sides.

§4.4.3. An inconsistency in the definition of ROTATE is repaired.

§4.6. The scaling convention for pixel arrays is relaxed.

§4.7. The operator FINDCOLOR is added.

§4.8.1. Errors in the definitions of LINETOX and LINETOY are repaired.

§4.8.2. The operators MASKVECTOR, MASKTRAPEZOIDX, and MASKTRAPEZOIDY are added. The action of MASKSTROKE is defined for degenerate trajectories.

§4.9.1. The *font* vector is changed to be a property vector.

§4.9.3. The operators MAKEVECLU and CONCAT are also allowed to appear in a metric master. The *characterMetrics* and *metrics* vectors are changed to be property vectors.

§4.10. The definition of correction operators is changed to allow printers more flexibility in implementation.

§5.1.1. "Levels" are renamed to "subsets." The subset structure is updated to reflect the new operators and the definition of the *Gray* enhancement is changed slightly.

Appendix B. Encoding values for REM, ROUND, and new operators are specified.

The following changes convert Interpress version 2.0 to version 2.1:

§2.4.6. The ERROR operator is added; MERGEPROP can be called from the master.

§2.5. The version number in the header is changed from 2.0 to 2.1.

§2.5.3. The encoding-notation for *sequenceString* is generalized and subordinated to the definition in the Xerox Character Code Standard.

§3.2. The notion of hierarchical name is replaced by more precise definitions of universal identifier, universal name, and environment name.

§3.3.3. Instructions *subset*, *environmentUses*, *insertFileUses*, and *stacking* are added.

§4.6. The definition of MAKEPIXELARRAY is generalized and the operator EXTRACTPIXELARRAY is defined.

§4.7. The notions of color operators and color model operators are introduced, as well as the operators FINDCOLOROPERATOR, FINDCOLORMODELOPERATOR, and MAKESAMPLEDCOLOR.

§4.10. The suggested implementation of *COMPUTECORRECTIONS is altered slightly.

§5.1. The set of enhancement modules is replaced by a second subset, the *reference subset*.

The following changes convert Interpress version 2.1 to version 3.0:

§2.2.1. Integer is renamed Cardinal and *maxInteger* is renamed *maxCardinal.*

§2.2.5. Operators obtained from the environment are identified as composed operators.

§2.3. Variables used by *DoMaster* and *DoBlock* are identified as part of the state.

§2.4. GETP is defined; the universal property vector definition is moved to §3.2.4.

§2.4.5. FINDOPERATOR is defined.

§2.4.7. The IFCOPY *testCopy* operator is restricted to BASE operators.

§2.5. PAGEINSTRUCTIONS are renamed CONTENTINSTRUCTIONS; the version number in the header is changed from 2.1 to 3.0.

§2.5.3. *S*equence types *sequenceCCITT-4PixelVector* and *sequenceInsertMaster* are defined. *sequenceInsertFile* is restricted to Bodies and precisely defined for a standard type of file.

§3.1. The skeleton structure is extended to include contents: page instructions are replaced by content instructions. *PGET, *PSET, *ISBODY, and *MERGECONTENTINSTRUCTIONS are defined. The skeleton processing program is replaced by *DoMaster*, *DoBlock*, and *DoMediumSet.*

§3.2. References to FINDOPERATOR and *sequenceInsertMaster* are added.

§3.2.1. Revised to describe both structured names and universal names.

§3.2.2. Operators are added to environment objects.

§3.2.3. Universal property vectors are defined (previously in §2.4.3).

§3.3. The policy of ignoring unrecognized printing instructions is stated.

§3.3.2. The break page discussion is moved to Appendix E.1.

§3.3.3. Special *default* identifiers are removed from instruction definitions. The discussion of run vectors is moved to §3.3.2. Printing instructions *breakPageFont, docComment, docCreation Date, docCreator, docName, environmentUses, finishing,* and *subset* are moved to Appendix E.3. Printing instructions *docPasswrod, jobSender, jobRecipient, jobStartMessage, jobEndMessage, jobStartWait, jobEndWait, jobAccount, jobPriority, jobSummary, jobErrorAbort,* and *jobPassword* are removed. Explicit default and priority behavior is given for instructions. The rotation for *duplex* is described in the *plex* instruction. The definition of the *media* instruction is revised. The *insertFileUses* instruction is split into *insertFileMapping* and *insertFileNames.* The *stacking* instruction is replaced by *outPosition* and moved to Appendix E.3.

§3.4. Page instructions are generalized to content instructions. The content instructions *contentInsertFileMapping, contentPageSelect,* and *contentPlex* are added.

§4.2. The imager variable *showVec* is renamed *font,* and its type is changed from Vector to Font. The imager variables *strokeJoint* and *clipper* are added.

§4.3.1. Setting the *medium-* or *field-* imager variables is prohibited. Conventions for placement of page images on two-sided media are specified.

§4.4.3. MAKET can be called from the master.

§4.4.6. is deleted; SHOW and SHOWANDXREL are moved to §4.9.

§4.7.1. The definition of MAKEGRAY is generalized.

§4.7.3. The operators SETSAMPLEDCOLOR and SETSAMPLEDBLACK are added.

§4.8. The *clipper* variable is added to the list of variables that control the operation of mask operators. Strokes are moved to a separate subsection, and the subsections are renumbered.

§4.8.1. The operators CURVETO, CONICTO, ARCTO, and MAKEOUTLINEODD are added.

§4.8.3. The action of MASKSTROKE is defined more precisely, and the effect of the *strokeJoint* variable is specified: MASKSTROKECLOSED is added. Dashed strokes are introduced: the operator MASKDASHEDSTROKE is added.

§4.8.5. Clipping is introduced; the operators CLIPOUTLINE and CLIPRECTANGLE are added.

§4.9. Character "widths" are renamed "escapements". The Font type is introduced, and the MAKEFONT operator is added. In a FontDescription, the *operators* property is replaced by the *characterMasks* property, and the *transformation* and *substituteIndex* properties are added; in a CharacterMetrics vector, the values of the *correction* property are rearranged. FontDescription and CharacterMetrics properties not needed by the imager are removed to the Xerox Font Interchange Standard. Conventions for the character coordinate system are clarified. Character operators are defined explicitly in terms of the information in a FontDescription. The MASKCHAR and *DOCHAR operators are added. SHOW and SHOWANDXREL are moved to this section, and SHOWANDFIXEDXREL is added.

§5.1. The subsets discussion is generalized to printer constraints. §5.1.1 is moved to §5.2.1-3 , and §5.1.2-4 moved to §5.4.

§5.2. "Subsets" are renamed to "Sets". The Text subset is expanded to become the "Commercial set", the Reference subset is expanded to become the "Publication set", and a "Professional Graphics set" is defined.

§5.3. Accessing the environment is discussed.

§5.4. Complexity issues are discussed (previously §5.1.2-4).

§5.5. Document handling and finishing is discussed.

§5.6. Numeric precision is discussed (previously §5.3). Minimum limits are defined. (Table 5.2).

§5.7. Error handling is discussed (previously §5.4).

§B.4 and §B.5 are removed.

§E. This appendix for printing instructions is added.

§E.2. Messages to the operator console are described and type *messageString* is defined.

§E.3. Explicit default and priority behavior are given for instructions. The *breakPageType*, *docStartMessage*, and *docEndMessage* instructions are added.

§E.4. The *contentOutputPosition* instruction is added.

E. PRINTING INSTRUCTIONS

Printing instructions provide a place to give information about the document and describe how the document should be printed. Printing instructions provide

- selection of which copies to print,
- specification of which pages are part of which copies,
- document finishing specifications,
- a description of supplemental pages such as the break page,
- information regarding resources required to print the master, and
- administrative information.

This appendix lists printing instructions which specify document finishing, specify information to be printed on supplemental pages such as the break page, and provide administrative information. Instructions which are involved in processing the skeleton are presented in §3.3 along with the general prescription for printing instruction management.

E.1 The break page

Several instructions provide specifications for and information to be printed on a break page which the printer may provide as a cover sheet and as a separator between successive documents. The break page may give the document name, creation date, printing date, recipient name, and messages describing errors, etc. Its layout is controlled by the printer.

Information that is supplied in instructions to be printed on the break page is contained in vectors of type *BreakPageString*. A BreakPageString is a Vector of Cardinal suitable for indexing the font specified by the *breakPageFont* instruction. This font indicates the character set of BreakPageStrings. The printer is not required to use this font, but is expected to print the strings in a font with similar symbols, so they are readable. Thus if character code 23 corresponds to "A" in this font, whenever 23 is encountered in BreakPageString, some sort of "A" must be printed. If the printer has no font with a character set that matches that of the font specified in the *breakPageFont* instruction, it may use a printer-dependent mechanism to print break page information.

Information for the break page may also be supplied by external sources. The character set appropriate to strings from external sources may depend upon the source and the printer, and is not specified by Interpress.

E.2 Message strings

Some instructions provide messages intended for a printer operator. Such messages can inform the operator of special handling required by the document (e.g., loading of special media). A message value is a vector of type *MessageString*. A MessageString is a Vector of Cardinal which encodes a message using character codes from the ISO 646 7-bit Coded Character Set for Information Processing Interchange.

Printers should print messages as soon as possible before (or in the case of *docEndMessage* after) imaging of the master. The actual method of message delivery is printer-dependent. Message instructions may be overriden by the printer if for some reason the message cannot be delivered (e.g., the printer does not provide an operator interface).

E.3 Standard instructions

The following instruction names and meanings are defined in this standard. It is not necessary for either the master or the external instructions to specify all of these instructions; default values are provided by *ADDINSTRUCTIONDEFAULTS. Each instruction description gives its name (i.e., the property name used in an instructions vector), followed by its type (i.e., the type which the value corresponding to the instruction must have). A master error results if the value specified for any of these instructions is not of the correct type. In the text describing each instruction, the symbol *value* denotes the value associated with the instruction property.

If an instruction value is specified both in the master and in external instructions, *ADDINSTRUCTIONDEFAULTS or *MERGECONTENTINSTRUCTIONS establishes the value which is used by precedence. In the following definitions, the parenthesized note after the definition of the value's type specifies whether the value computed in the master (Master) or the value supplied by external instruction (External) is given precedence.

breakPageFont: Vector of Identifier. (Master)

The universal name of a font to be used to print BreakPageStrings. The font will be obtained by <*value* FINDFONT>. Default: a font chosen by the printer.

breakPageType: Identifier or Vector of Identifier. (External)

This instruction identifies the form of the break page. The printer may override this instruction if a particular break page type is required. The possible values of this instruction are:

none. No break page is printed.

terse. A short form for the break page which contains only minimal information.

verbose. A break page containing maximal information about the job.

Vector of Identifier. A universal name (§3.2.2) specifies a special break page format.

Default: a break page form chosen by the printer.

docComment: BreakPageString. (Master)
A comment to be printed on the break page.

docCreation Date: BreakPageString. (Master)
A string to be printed on the break page that shows the date and time when the document was created.

docCreator: BreakPageString. (Master)
A string to be printed on the break page that shows the name of the person who created the document.

docName: BreakPageString. (Master)
A string to be printed on the break page containing an identifying name for the document.

docEndMessage: MessageString. (Master)
A message to be delivered to the operator as soon as possible after master imaging.

docStartMessage: MessageString, (Master)
A message to be delivered to the operator prior to and as close as possible to the time of imaging the master.

finishing: Identifier or Vector of Identifier. (External)
This instruction specifies the name of a document-finishing technique to be applied. The values of this instruction are:

none: No finishing is done.

cornerStaple: A single staple is inserted in each copy in the upper-left corner, i.e., near the $x=0$, $y=mediumYSize$ point of the Interpress coordinate system of the page with $pageNumber=1$ (§4.3.1, §3.1). The break page, if any, may be stapled to a copy.

Vector of Identifier: A universal name (§3.2.2) may also appear as the value of a *finishing* instruction to define printer-dependent finishing, e.g., [*exxon, bind*].

Default: a printer-dependent default action is taken.

outputPosition: Run of Cardinal (External)
The run vector is indexed by copy number and the value is a positive integer indicating the output position for the copy. The printer makes its best effort at implementing output positions by means such as stacking offsets or sorting bins. Default: [10^7,1].

set: Identifier or Vector of Identifier. (External)
This instruction provides an indication of the set of Interpress types and primitives required to execute the master (§5.1). This set includes all types and primitives used by composed operators created by the master, regardless of whether the operators are executed by the master itself. The scope of this instruction includes bodies inserted directly or indirectly by *sequenceInsertFile* tokens and masters included by means of *sequenceInsertMaster* tokens. The printer is entitled to provide only the capabilities of the specified set for processing the master. The possible values of this instruction are:

professionalGraphics: The master is assumed to use any of the facilities of the professional graphics set.

publication: The master is assumed to use only the facilities of the publication set.

commercial: The master is assumed to use only the facilities of the commercial set.

Vector of Identifier: A universal name (§3.2.2) may also appear as the subset definition, e.g., [*xerox, performance*].

Default: the master is presumed to use all of the facilities of the language.

The following instructions provide *hints* about printer capabilities and objects in the printer's environment required to execute the master, in the sense that the information provided is not required to be complete or accurate. The scope of these instructions includes bodies inserted directly or indirectly by *sequenceInsertFile* tokens and masters included by means of *sequenceInsertMaster* tokens. It also includes composed operators created by the master, regardless of whether the operators are executed by the master itself. If these hints are not provided, it is assumed that the master may use any of the objects in the printer's environment.

environmentNames: Vector of Vector Identifier (External)
Each Vector of Identifiers is an environment name (§3.2.3). This instruction provides a hint as to the names of objects in the environment that the master uses.

For example, the value [[*colors, xerox, highlight*], [*fonts, xerox, xc2-0-0, times*], [*fonts, xerox, xc2-0-0, helvetica*]] indicates that the master obtains a color by calling <[*xerox, highlight*]] FINDCOLOR>, and two fonts by calling <[*xerox, xc2-0-0, times*] FINDFONT> and <[*xerox, xc2-0-0, helvetica*] FINDFONT>.

insertFileNames: Identifier or Vector of Vector of Cardinal. (Master)
This instruction provides a hint as to the names of files used with the *sequenceInsertFile* or *sequenceInsertMaster* encoding-notations (§2.5.3). Each element is a Vector of Cardinal that is a copy of the data bytes of a *sequenceInsertFile* or *sequenceInsertMaster* token that appears in the master. Default: the master is presumed to access any files in the printer's environment.

pixelArrayTransformationUses: Vector of Transformation. (External)
This vector provides a hint as to the resolution and orientation of the pixel arrays that this master images. Each element in the vector is a transformation from the pixel array coordinate system to the Interpress coordinate system. Default: the master is presumed to use any pixel array transformations.

E.4 Content instructions

Content instructions which provide administrative and finishing information are presented below. Other content instructions which are involved in processing the skeleton are described in §3.3.4. Content instructions do not have default values; if they are absent, the default behavior is described by the master's instructionsBody. Integration of a content instruction into the current instructions set is done by *MERGECONTENTINSTRUCTIONS according to the prescription provided in the instruction's description. Further information on content instruction handling is provided in §3.3.4 and in the program in §3.1.

contentOutputPosition: Run of Cardinal. (External)

This instruction selects an output position for pages generated by the current content. The run vector is indexed by copy number and the value is a positive integer indicating the output position. The printer will make its best effort at implementing output positions by means such as stacking offsets or sorting bins. The value takes precedence over the master's *outputPosition* instruction. If an *outputPosition* instruction is present in the external instructions, it takes precedence over both the master's *outputPosition* and *contentOutputPosition*. In *MERGECONTENTINSTRUCTIONS, the new value of this instruction replaces any previous value for the scope of the content.

An italicized word in a definition is indexed and defined in this glossary. The parenthesized number at the end of each definition is the section in which the term is introduced.

amplifying characters: characters whose *escapement* can be easily modified to achieve *justification* (4.9)

appearance error: an error in the appearance of the *page image*, usually because the *master* invokes a function that the *imager* cannot accommodate (5.7)

approximation: finding an *external* color or font which is close to the one requested, but not necessarily identical (4.7.1, 4.9.2)

argument: a value popped from the *stack* by the execution of an *operator* (2.4)

base language: the syntax and semantic framework of Interpress, without any *primitive operators* whose primary use is to generate *output* (2)

baseline: in Latin alphabets, a horizontal line just under the "bottom" of non-descending characters (4.9)

body: a sequence of *literals* bracketed by { and }, which can be used to form the executable part of a *composed operator* (2.2.5)

body operator: *a primitive operator* which takes a *body* as its last *argument* (2.2.5)

break page: a page automatically printed at the beginning of a job to identify the output of the job and to separate it from that of adjacent jobs (E.1)

cardinal: a nonnegative mathematical integer in a limited range: one of the types of the *base language* (2.2.1)

char: abbreviation for character

character coordinate system: a standard coordinate system in which each *character operator* is defined (4.9)

character index: a Cardinal, sometimes called a "character code," that identifies a particular character: used to index a *font* (4.9)

character operator: a *composed operator* which, when executed, defines a character's *mask, escapement,* and spacing *correction* (4.9)

co: abbreviation for *composed operator*

color: the specification of the color with which to show a primitve stage (4.1, 4.7)

color operator: an *operator* that converts color coordinates into an Interpress *color* (4.7.1)

color model operator: an *operator* that constructs a *color operator* according to a particular color model (4.7.1)

composed operator: an *operator* defined in the *master* (2.2.5)

compression: a computation that reduces the number of bits required to specify some data, usually a *pixel array* (4.6)

context: a particular execution of a *composed operator* (2.4.2)

convenience operator: a *redundant operator*, usually introduced to reduce the number of steps in a frequently-occurring sequence

coordinate system: conventions used to describe locations on a two-dimensional surface (4.3)

correct: to compensate for differences between the actual *escapements* of *character operators* used by the *printer* and those assumed by the *creator* (4.10)

creator: the person or program which constructs an Interpress *master* (1)

current position: a point on the *page image*, often used to indicate where the *origin* of the next character should be placed (4.5)

current transformation: a *transformation* that converts from *master coordinates* to *device coordinates* (4.4)

DCS: abbreviation for *device coordinate system*

decompression: expanding *compressed* data into its original form (4.6)

device coordinate system: a device-dependent *coordinate system* suitable for driving the printing device (4.3)

device-independent: does not depend on properties of the printing device (4.3)

duplex: a mode of printing in which images are placed on both sides of a sheet of paper: also a *printing instruction* (3.3.3)

element: one of the values which make up a *vector* (2.2.4)

encoding: a particular representation of Interpress *masters* (2.5)

environment: the set of objects made available to a *master* by a *printer*, e.g., *fonts, colors, decompresssion operators* (3.2)

environment name: the unique name of an object in the printer's *environment*; a *vector* of *identifiers* in which the first identifier defines the type of the object and the remaining identifiers are the *universal name* of the object (3.2.2)

escapement: of a character, the spacing from one character to the next (4.9)

external instructions: those *printing instructions* that are supplied by mechanisms outside an Interpress master (3)

external value: a value not defined in the master, but obtained from the printer by a FIND operator (3.2)

f: abbreviation for *frame*

font: a collection of character definitions (4.9)

frame: a vector associated with an execution of a *composed operator* (2.3.2)

good image: an image specified with just sufficient precision to match the *ideal image* (4.3.4)

grid points: a grid overlaid on the *device coordinate system* for describing the spatial resolution of the printing device (4.3.4)

header: an identifying string at the beginning of an *encoded* Interpress master (2.5)

hierarchical: a tree-structured naming system, in which each name is a sequence of simple names which traces out a path from the root of the tree (3.2.1)

hierarchical name: a *vector* of *identifiers* that represent a structured name (3.2.1)

ICS: abbreviation for *Interpress coordinate system*

id: abbreviation for *identifier*

ideal image: an image that results from ideal (infinite) precision interpretation of arithmetic and imaging *operators* (4.3.4)

identifier: a sequence of characters normally used to name an external value: one of the types of the *base language* (2.2.2)

imager: the software module that interprets imaging operators to build *page images* (4.1)

imager state: twenty-three *variables* that control the functioning of many imager *operators* (4.2)

imaging model: the process whereby primitive images specified by a *color* and a *mask* are built up on a *page image* (4.1)

initial frame: a *vector* which is part of a *composed operator* and used to initialize the *frame* for each execution of the operator (2.4.2)

ins: abbreviation for *instructions*

instance: usually refers to an image on the page of standard *symbol*. For example, the word SHIPS when printed, contains two instances of the symbol S (4.4.1)

instructions: abbreviation for *printing instructions*

instuctions body: an optional body in a *master* that contains *printing instructions* (3)

Interpress coordinate system: a device-independent *coordinate system* for specifying locations on the *page image* (4.3)

justify: to space characters out so that they completely fill a pre-determined region, such as the space between margins (4.9)

kern: the portion of a typeface that projects beyond the body or shank of a character

ligature: a character or type combining two or more letters, such as *fi*

limit: a restriction on the size of some object (5.6.1)

limited imager: an *imager* that cannot handle pages of arbitrary complexity (5.4)

literal: a representation in a master of value (2.2)

lower bound: the integer which names the first *element* of a *vector* (2.2.4)

mark: a special value which can only be popped from the *stack* by certain *operators* (2.2.3)

mark recovery: an error recovery procedure which pops the *stack* to the topmost *mark* and finds a *matching* point in *operator* execution (2.4.1)

mask: a description of the shape of a primitive image that will be added to the page image (4.1, 4.8)

master: an Interpress program (1)

master coordinates: coordinate information specified by the master as arguments to *imaging* operators (4.3)

master error: the result of executing a *primitive* without meeting the conditions stated in its definition (2.4.1, 5.3)

matching: an UNMARK or COUNT operator executed in the same *context* as the MARK which pushed a particular *mark* value onto the *stack* (2.2.3)

matrix: a representation of a *transformation* (4.4)

medium: the identity of the material on which a *page image* is printed (3, 4.2)

metrics: of a character or *font*, the measurements of its critical dimensions (4.9)

name: a *cardinal* or *identifier* used to specify an *element* of a *vector* (2.2.4, 3.2)

net transformation: the total transformation from a pixel array's or font operator's *standard coordinate system* to the *Interpress coordinate system* (4.6)

normal viewing orientation: the standard orientation of a page (or other form of image output) (4.3)

number: a rational number in a particular subset: one of the types of the *base language* (2.2.1)

op: abbreviation for *operator*

operator: a value which can be executed to cause *state* changes and *output* (2.2.5)

operator restrictions: rules limiting the *primitives* which can be executed in various parts of the master (3.1.1)

origin: a reference point on a character mask (4.9)

outline: a set of closed *trajectories*, usually used to define the outline of a region (4.8.1)

output: result of executing a *master* (2.3)

page: a unit of *output* (3)

page image body: the portion of a master which generates the *output* for a *page* (3.1)

page image: the image built by a *page image body*, which will be printed (4.1)

page instructions body: an optional portion of a master which specifies *printing instructions* for a particular page (3.3)

persistent: a *variable* whose value is not reset by a DOSAVE (2.3.3)

pixel: an element of a *pixel array* (4.6)

pixel array: a two-dimensional array of *samples* that define the *color* everywhere in a rectangular region (4.6)

pixel array coordinate system: a standard coordinate system in which a rectangular array of *samples* is defined (4.6)

point: a printer's unit of distance, roughly 1/72 inch

preamble: a part of the *skeleton* which establishes the *initial frame* for execution of the *page bodies* (3.1)

primitive: an *operator* built into Interpress and defined in the standard (2.2.5)

printer: a device which accepts Interpress *masters* and produces the corresponding *images*

printer-dependent: a part of Interpress whose detailed interpretation is not standardized, but instead left to individual printer manufacturers or operators to specify

printing instructions: commands that control the printing of an Interpress *master* (3.3)

priority: the property that determines which of two overlapping primitive images will appear to be "on top" (4.1)

property name: an *identifier* used in a *property vector* to name a corresponding value (2.4.3)

property vector: a *vector* formatted so as to describe (*property name*, value) pairs (2.4.3)

raster, raster-scan: a two-dimensional array of *pixels* that covers an image, and the process of methodically scanning past each pixel on the image

redundant operator: a *primitive operator* that is an abbreviation for a simple Interpress program

result: a value pushed onto the *stack* and left there by the execution of an *operator* (2.4)

registry: a set of *identifiers* which controls a particular point in a *hierarchical name* space (3.2)

rounding: usually, finding the *grid point* closest to a *device coordinate* (4.3, 4.12)

sample: a record of the *color* at a *pixel*, i.e., a point in an image (4.6)

scan-conversion: the act of converting geometric or *sampled* intensity information into a *raster-scanned* image (4.8)

scanned-image: see *pixel array*

set: a characterization of the capabilities of an Interpress *printer* (5.2)

simplex: a mode of printing in which an image is placed on only one side of each sheet of paper; also a *printing instruction* (3.3.3)

skeleton: the global structure of a *master*, down to the level of the outermost *bodies* (3.1)

spaceband character: a character whose width is expanded by the factor *amplifySpace* (4.9)

spot: a small region of the output image whose color can be controlled by the printing device independently of all other regions (4.3, 4.6)

stack: a last-in first-out sequence of values used to communicate information between *operator* executions (2.3.1)

standard coordinate system: the system in which a pixel array or font is defined (4.6, 4.9)

state: the information which can affect further execution of a master (2.3)

stroke: a *mask* obtained by broadening a *trajectory* or *outline* to have uniform width (4.8.3)

symbol: a graphical shape; several *instances* of a symbol may appear in a *page image* (4.4.1)

T_{ID}: the *transformation* from the *Interpress coordinate system* to the *device coordinate system* (4.3.5)

token: a primitive element of an Interpress master (2.5)

trajectory: a set of connected lines used to determine where *strokes* should be drawn (4.8.1)

transformation: a conversion of coordinate information from one *coordinate system* to another (4.4)

transition function: a mapping from *states* into states and *output*, which defines the meaning of an *operator* (2.2.5)

type: one of the classes of values (2.2)

universal identifier: an *identifier* defined in the Interpress *universal registry* (3.2.1)

universal name: a *vector* of *identifiers* in which the first identifier is a *universal identifier* (3.2.1)

universal property vector: a *property vector* that can be extended using property names that are *universal names* (3.2.3)

universal registry: a registry of unique *identifiers* assigned to organizations that wish

to create *structured names* of objects in a printer's *environment* (3.2.1, C)

unlimited imager: an *imager* that can print pages of arbitrary complexity (5.4)

upper bound: the integer name of the last *element* of a *vector* (2.2.4)

variable: part of the imager state (4.2)

vec: abbreviation for *vector*

vector: a sequence of values named by Integers (2.2.4)

XEROX

Xerox Corporation
Stamford, Connecticut 06904

Printed in U.S.A.

610P72582A