XEROX

Xerox System Integration Standard

Interpress Electronic Printing Standard

Xerox Private Data

XEROX Interpress Electronic Printing Standard Xerox System Integration Standard XSIS 048306

XEROX

Xerox System Integration Standard

Interpress Electronic Printing Standard

Version 2.0
XSIS 048306
June 1983

Xerox Corporation
Stamford, Connecticut 06904

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. Significant differences between Interpress 82 and the current standard are summarized in Appendix D.

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

Figures

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 describes precisely the desired appearance of *a page 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. The *only* adjustments made by the printer are minor corrections to the positions of characters on a line, so as to present a line of text which has been adjusted to the actual font widths and resolution of that printer.

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 are used to describe the kinds of values required by certain operators:

- *Any* is a type which accepts a value of any type except Body or Mark.
- *Integer* 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 in § 2.2 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 § 5.1.1.

2.2.1 Numbers and Integers

A Number is an element of a certain subset of the mathematical rationals. This subset includes every rational which can be represented in the form $i \times 2^e$, where i is an integer in the range $-(2^{24}-1)\,..\,(2^{24}-1)$, and e is an integer in the range $-100\,..\,100$; it may include other rationals as well.

Throughout this specification, "integer" means either a mathematical integer or a Number which is a mathematical integer. An Integer (capitalized) is an integer in the range 0..*maxInteger;* the Integers are a subset of the Numbers and not a completely distinct type. An Integer 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. The value it represents is the Number value closest to this rational number; if two Numbers are equally close, the smaller is chosen.

2.2.2 Identifiers

An Identifier 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 Integers. Its indexing information is two integers 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 ith element in the Vector is named by the Integer $l+i-1$. 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 a * 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 the *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. There are no composed operator literals.

Composed operators 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 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 and IFELSE operators, which take a Body and an Integer as arguments and execute the Body if the Integer is non-zero. For example, 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 up 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
} MAKESIMPLECO


{-- 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
    MASKSTROKE
} MAKESIMPLECO
```

2.3 State

The alterable state of the machine that executes Interpress masters consists of the *stack* (§ 2.3.1), the *contexts* of composed operators in the midst of execution (§ 2.3.2), and 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.

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 chapter 4.

2.3.2 Frames and contexts†

A composed operator is constructed from a Body and a Vector which is the *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*. The frame is initialized to the value of the 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 the initial frame) some global values available when it is defined. Changes to locals 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 state transition function of an operator *Op* is defined by text which begins:

$\langle a_1: T_1\rangle \ldots \langle a_n: T_n\rangle\ Op \rightarrow \langle r_1: U_1\rangle \ldots \langle r_m: U_m\rangle$
where . . .

in which the T's and U's are types. If r_i is the same as a_j, then 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 pidgin 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; hence [] 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 an Integer 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 hierarchical 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 will 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; otherwise, composed operators in execution are exited until the one which placed the mark is executing; and (c) literals are skipped in this operator until an UNMARK0 literal is found. The UNMARK0 found in step (c) is 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 (c) 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†

A composed operator 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 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†

$\langle v$: Vector$\rangle$ $\langle j$: Integer$\rangle$ GET $\rightarrow$ $\langle x$: Any$\rangle$

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.

$\langle x_1$: Any$\rangle$...$\langle x_n$: Any$\rangle$ $\langle l$: Integer$\rangle$ $\langle u$: Integer$\rangle$ MAKEVECLU $\rightarrow$ $\langle v$: Vector$\rangle$

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 $x_n, \ldots, x_1$, 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., $\langle v\ l+i-1$ GET$\rangle = x_i$. A master error occurs unless $0 \leq n \leq maxVecSize$.

$\langle x_1$: Any$\rangle$...$\langle x_n$: Any$\rangle$ $\langle n$: Integer$\rangle$ MAKEVEC $\rightarrow$ $\langle v$: Vector$\rangle$

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., $\langle v\ i-1$ GET $\rangle = x_i$. A master error occurs unless $n \leq maxVecSize$. § 2.4 describes a notation for writing calls of MAKEVEC in examples.

$\langle v$: Vector$\rangle$ SHAPE $\rightarrow$ $\langle l$: Integer$\rangle$ $\langle n$: Integer$\rangle$

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 identifiers and vectors of identifiers be used as property names; other values are permitted, but are not found by GETPROP. The following operators are defined for property vectors:

$\langle v$: Vector$\rangle$ $\langle propName$: Any$\rangle$ GETPROP $\rightarrow$ $\langle value$: Any$\rangle$ $\langle 1$: Integer$\rangle$ or $\rightarrow$ $\langle 0$: Integer$\rangle$

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 $\langle v\ i$ GET *propName* *EQN$\rangle=1$. *EQN compares Identifiers, Numbers, and Vectors for equality; it is defined precisely in § 2.4.8. If no match is found, 0 is returned on the stack. If a match is found, $\langle v\ i$ 1 ADD GET 1$\rangle$ is executed to place on the stack the property's value and the Integer 1, which indicates that a match has been found. A master error occurs if $(u-l+1)$ MOD $2 \neq 0$.

$\langle v_1$: Vector$\rangle$ $\langle v_2$: Vector$\rangle$ *MERGEPROP $\rightarrow$ $\langle v_3$: Vector$\rangle$

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, $\langle v_3\ n$ GETPROP$\rangle$ is equivalent to $\langle v_2\ n$ GETPROP DUP NOT { POP $v_1\ n$ GETPROP } IF$\rangle$.

Note that this definition does not fully specify the order of elements in v_3, but does fully define the behavior of v_3 when accessed with GETPROP.

A *universal property vector* is a form of property vector that may contain both property names standardized by Interpress and hierarchical names (§ 3.2.2) that allow growth of the name space. A property name in a universal property vector must be either:

- An Identifier from a set defined by Interpress for this purpose.
- A Vector of Identifiers, which specifies a hierarchical property name. The first Identifier in the Vector is a universal name (§ 3.2.2).

2.4.4 Frame operators†

$\langle j$: Integer$\rangle$ FGET $\rightarrow \langle x$: Any$\rangle$

where x is the current value of the jth element of the frame. A master error occurs unless $j <$ *topFrameSize*. The value of *topFrameSize* may be limited (§ 5.1.1).

$\langle x$: Any$\rangle$ $\langle j$: Integer$\rangle$ FSET $\rightarrow$ $\langle\rangle$

where the value of the frame element named by j, becomes x. A master error occurs unless $j <$ *topFrameSize*.

2.4.5 Operator operators†

$\langle b$: Body$\rangle$ MAKESIMPLECO $\rightarrow$ $\langle o$: Operator$\rangle$

where o is a composed operator which has body b and initial frame equal to the value of the frame when MAKESIMPLECO is executed.

$\langle o$: Operator$\rangle$ DO $\rightarrow$ -- 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).

$\langle o$: Operator$\rangle$ DOSAVE $\rightarrow$ -- 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 imager variables which are not persistent to their values just before the DOSAVE.

$\langle o$: Operator$\rangle$ DOSAVEALL $\rightarrow$ -- 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 imager variables to their values just before the DOSAVEALL.

$\langle b$: Body$\rangle$ DOSAVESIMPLEBODY $\rightarrow$ -- the effect on the stack depends on b --

where the effect is b MAKESIMPLECO DOSAVE.

2.4.6 Stack operators†

$\langle x$: Any$\rangle$ POP $\rightarrow$ $\langle\rangle$

where the top element of the stack is removed with no other effects.

$\langle x_1$: Any$\rangle$...$\langle x_{depth}$: Any$\rangle$ $\langle depth$: Integer$\rangle$ COPY $\rightarrow$ $\langle x_1\rangle$...$\langle x_{depth}\rangle$ $\langle x_1\rangle$...$\langle x_{depth}\rangle$

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.

$\langle x$: Any$\rangle$ DUP $\rightarrow$ $\langle x\rangle$ $\langle x\rangle$

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

$\langle x_1$: Any$\rangle$...$\langle x_{depth}$: Any$\rangle$ $\langle depth$: Integer$\rangle$ $\langle moveFirst$: Integer$\rangle$ ROLL

$\rightarrow$ $\langle x_{moveFirst+1}\rangle$...$\langle x_{depth}\rangle$ $\langle x_1\rangle$...$\langle x_{moveFirst}\rangle$

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

$\langle x$: Any$\rangle$ $\langle y$: Any$\rangle$ EXCH $\rightarrow$ $\langle y\rangle$ $\langle x\rangle$

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

$\langle x_1$: Any$\rangle$...$\langle x_n$: Any$\rangle$ $\langle n$: Integer$\rangle$ MARK$\rightarrow$ $\langle m$: Mark$\rangle$ $\langle x_1\rangle$...$\langle x_n\rangle$

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.

$\langle m$: Mark$\rangle$ $\langle x_1$: Any$\rangle$...$\langle x_n$: Any$\rangle$ $\langle n$: Integer$\rangle$ UNMARK $\rightarrow$ $\langle x_1\rangle$...$\langle x_n\rangle$

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.

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:* Integer⟩
: **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.

2.4.7 Control operators†

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

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

Note that *i* is not on the stack when the body is executed. The effect of "**if** *i* **then** 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, and *b* is executed unless *testCopy* returns 0. 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 Integer; 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:* Integer⟩
: **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:* Integer⟩
    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: Integer;
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:* Integer⟩
    where $c=1$ if $a>b$, $c=0$ otherwise.

⟨*a:* Number⟩ ⟨*b:* Number⟩ GE → ⟨*c:* Integer⟩
    where $c=1$ if $a\geq b$, $c=0$ otherwise.

⟨*a:* Integer⟩ ⟨*b:* Integer⟩ AND → ⟨*c:* Integer⟩
    where $c=1$ if $a\neq 0$ and $b\neq 0$, $c=0$ otherwise.

⟨*a:* Integer⟩ ⟨*b:* Integer⟩ OR → ⟨*c:* Integer⟩
    where $c=1$ if $a\neq 0$ or $b\neq 0$, $c=0$ otherwise.

⟨*b:* Integer⟩ NOT → ⟨*c:* Integer⟩
    where $c=1$ if $b=0$, $c=0$ otherwise.

⟨*a:* Any⟩ TYPE → ⟨*c:* Integer⟩
    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 (mathematical) integer $\leq a$.

⟨*a:* Number⟩ CEILING → ⟨*c:* Number⟩
    where *c* is the least (mathematical) integer $\geq a$.

⟨*a:* Number⟩ TRUNC → ⟨*c:* Number⟩
    where *c* is the (mathematical) 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.

⟨*a:* Number⟩ ⟨*b:* Number⟩ MOD → ⟨*c:* Number⟩
  where $c = a - b \times \mathrm{FLOOR}(a/b)$; $b \neq 0$ is required.

⟨*a:* Number⟩ ⟨*b:* Number⟩ REM → ⟨*c:* Number⟩
  where $c = a - b \times \mathrm{TRUNC}(a/b)$; $b \neq 0$ 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, PAGEINSTRUCTIONS, "{", 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/2.0□`" 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 (2/0) 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(112);   -- p --
   . . .
AppendByte(120);   -- x --    AppendByte(47);    -- / --    AppendByte(50);    -- 2 --
AppendByte(46);    -- . --    AppendByte(48);    -- 0 --    AppendByte(32);    -- □ --
```

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

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.

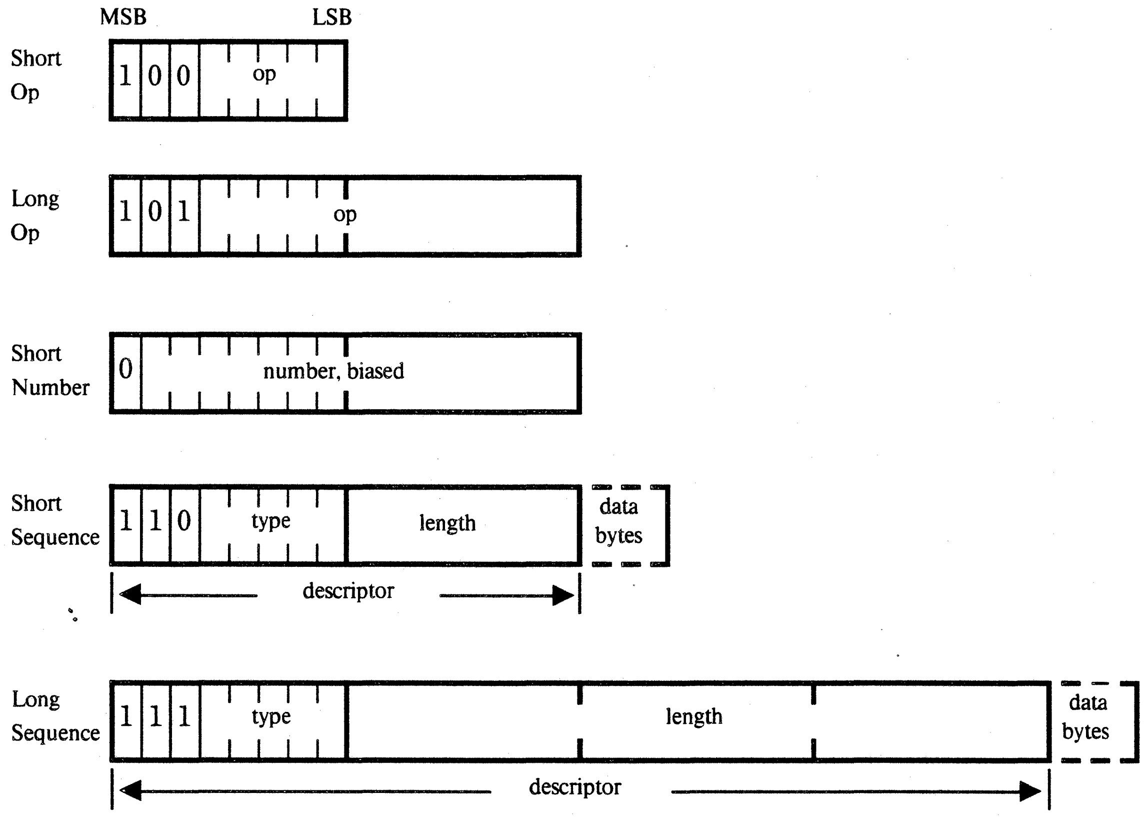


Figure 2.1 Token formats.

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 byteth 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 up 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 Integers are a subset of Numbers and not a totally distinct type, it is not compulsory to encode an Integer 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.

- 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 $-(256^i \text{ div } 2) \le n$ and $n < (256^i \text{ div } 2)$ then done := true end;
    BytesInInt := i
    end;

procedure AppendInteger(n: Number);
    const i = BytesInInt(n);
    begin
    if $-4000 \le n$ and $n \le 28767$ 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 is a token which encodes the "}" symbol, generated with *AppendOp*(107).

BEGIN, END, and PAGEINSTRUCTIONS. 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 Integers, in which each element lies in the range 0..65534. These vectors occur especially frequently as arguments to the SHOW operator. If all the elements are in the range 0..254, the vector is encoded as a sequence of type *sequenceString*, with *length* equal to the number of elements in the vector. Such a token is equivalent to

a sequence of Integer literals, one for each data byte, in the order in which bytes appear in the *sequenceString;*

followed by an Integer literal with the value *length;*

followed by the Operator literal MAKEVEC.

The result is thus a vector with $l=0$ and $u=length-1$ whose elements are the data bytes of the *sequenceString.* For example, an encoding of 12 13 202 3 MAKEVEC has exactly the same effect as:

AppendSequenceDescriptor(*sequenceString*, 3); *AppendByte*(12); *AppendByte*(13); *AppendByte*(202);

To represent vector elements in the range 256..65534, an *escape* mechanism is used. The mechanism cannot encode i if i **mod** 256=255; hence a vector with such an element cannot be encoded as a string. A byte in the sequence with value 255 is an escape byte; the byte immediately following it specifies an *offset;* thereafter *offset**256 is added to all subsequent bytes in the sequence until the escape once again changes the offset. The offset is set to zero at the beginning of the decoding. The *AppendString* procedure shows how a vector might be encoded. 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: Integer);
    var byteCount: Integer := 0; offset: Integer;
    begin
    for pass := 1 to 2 do begin
        offset := 0;
        if pass=2 then AppendSequenceDescriptor(sequenceString, byteCount);
        for i := 0 to numElements-1 do begin
            if v[i]<0 or v[i]>65535 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 used to represent each number; call this b. The remaining data bytes are grouped into b byte parts, and each of these parts is a twos-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: Integer);
    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
```

Xerox Private Data

Pixel vectors. There are two types of sequence 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 Integers *v* formed from the data bytes and executing <[*Xerox, packed*] FINDDECOMPRESSOR DO>.

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

In both 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 either *sequencePackedPixelVector* or *sequenceCompressedPixelVector*.

Inserting from a file. There is a special encoding-notation which takes a file name as parameter, and is equivalent to inserting the contents of the file in the master, in place of the encoding-notation. Sequence type *sequenceInsertFile* has data bytes which are interpreted as character codes for the name of a file, in a printer-dependent character set. The *sequenceInsertFile* token is replaced by the tokens in the specified file. File insertions may nest to a depth of 16.

Any kind of printer-dependent processing may be done on the file to obtain a sequence of tokens; e.g., if an encoding header is present, that header and the BEGIN and END tokens might be stripped off, so that a file representing a complete master can also be inserted into another master. In fact, the inserted file need not actually contain tokens in the encoding; it might contain some representation compiled for a particular printer, as long as the net effect is the same as that produced by some sequence of tokens.

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
PAGEINSTRUCTIONS	105
"{"	106
"}"	107

Table 2.2 Values for sequence types

Name	Value (decimal)
sequenceComment	6
sequenceCompressedPixelVector	10
sequenceContinued	7
sequenceIdentifier	5
sequenceInsertfile	11
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 out of a sequence of bodies called its *skeleton.* It also explains the conventions for naming external objects such as fonts and for passing special instructions to an Interpress printer.

3.1 The skeleton

A master consists of a sequence of Bodies (§ 2.2.5) called the *skeleton.* Within each body, the rules of the base language prevail. The way in which the bodies are executed is outlined first and then described in precise detail.

The master contains:

- An optional *instructions body* that is executed to tell the printer how the master is to be printed.
- A body called the *preamble,* which establishes the initial frame for all the remaining bodies.
- An arbitrary number of page descriptions. A page description may be a single *page image body,* which is executed to generate the image for the corresponding page. Alternatively, a page description may contain two bodies, a *page instructions body* followed by a *page image body.* The page instructions body determines any special printing instructions that are to apply to this page only. A special token in the skeleton distinguishes between the two kinds of page descriptions.

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, 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 a complete set of instructions. The details of printing instructions are given in § 3.3.

After the instructions body, if any, is executed, the preamble is executed with an initial frame containing *topFrameSize* zeros, and no arguments. The preamble returns no results, but the value of its frame after its execution is used as the initial frame for each page body. Thus the preamble can set up fonts, define composed operators, establish various parameters, etc., and leave all this information in the frame for the use of the page bodies.

Then each page description is processed in turn. If a page instructions body is present, it is executed to obtain an instructions vector that will apply for this page only. Then the page image body is executed, with the initial frame supplied by the preamble, and no arguments. A fresh printing surface is supplied to hold the image produced by each page body.

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, so that 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, and its entire effect is therefore captured in the output it produces.

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

```
skeleton              ::= instructionsBody block | block
block                 ::= BEGIN preambleBody pages END
pages                 ::= page | page pages
page                  ::= pageImageBody |
                          PAGEINSTRUCTIONS pageInstructionsBody pageImageBody
instructionsBody      ::= body
preambleBody          ::= body
pageImageBody         ::= body
pageInstructionsBody  ::= body
body                  ::= as defined in § 2.2.5
```

An example of a skeleton for a two-page master is { instructionsBody } BEGIN { preambleBody } { pageImageBody } PAGEINSTRUCTIONS { pageInstructionsBody } { pageImageBody } END.

The meaning of a master (i.e., the output it generates) is defined by the following pidgin Pascal program which executes the skeleton. The program intermixes Interpress operators with Pascal in an obvious way; it treats the Pascal variables as elements of an Interpress frame, which it can reference with FGET and set with FSET. It also uses several special operators which do not exist in true Interpress.

⟨*f:* Vector⟩ ⟨*b:* Body⟩ *MAKECOWITHFRAME returns a composed operator with body *b* and initial frame *f*.

*LASTFRAME returns the final value of the frame for the most recently completed execution of a composed operator.

⟨*m:* Vector⟩ ⟨*pageNumber:* Integer⟩ ⟨*duplex:* Integer⟩ ⟨*xImageShift:* Number⟩ *SETMEDIUM starts a new page. This operator is defined fully in § 4.2.

*OBTAINEXTERNALINSTRUCTIONS returns a property vector representing printing instructions supplied to the printer from outside.

<*computedInstructions:* Vector> <*externalInstructions:* Vector> *ADDINSTRUCTIONDEFAULTS returns a single property vector in which *computedInstructions* has been altered to include defaults specified by Interpress and printer-dependent defaults or constraints. A printer may implement *ADDINSTRUCTIONDEFAULTS in such a way that certain instructions present in *externalInstructions* will also be present in the result (e.g., passwords).

*RUNSIZE and *RUNGET are defined in § 3.3.3.

In the following program, the *instructionsBody* variable contains the instructions body or a null body, { }, if the master has no instructions body. The *preambleBody* variable contains the preamble. The *pageImageBodies* variable is assumed to contain a vector of page image bodies contained between the bracketing BEGIN and END. The *pageInstructionsBodies* variable contains a corresponding vector of page instruction bodies, with null bodies for those pages that do not have instructions. The program clears the initial frame vector *iFrame* to 0 and executes the preamble with DOSAVEALL, saving the final value of the frame in *iFrame*. Then it executes each page body, after executing *SETMEDIUM to set up the page.

```
instructionsBody, preambleBody: Body;
pageImageBodies, pageInstructionsBodies: Vector [1 .. numPages] of Body;
instructions: Vector of Any; -- final printing instructions --
mediaI, copySelectI, pageSelectI, onSimplexI, mediaSelectI, copyNameI: Vector of Any;
copyName: Identifier; copyNumber, pageNumber, i: Integer;
iFrame: Vector [0 .. topFrameSize−1] of Any;
-- 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 --
    *OBTAINEXTERNALINSTRUCTIONS -- push property vector on stack --
    iFrame FGET
    instructionsBody FGET
    *MAKECOWITHFRAME DOSAVEALL -- execute instructions-body --
    while COUNT>1 do { *MERGEPROP } -- merge all instructions --
    instructions FSET
UNMARK0
instructions FGET
*OBTAINEXTERNALINSTRUCTIONS
*ADDINSTRUCTIONDEFAULTS -- install Interpress and printer-dependent defaults --
instructions FSET -- and save as final printing instructions --
instructions FGET media GETPROP POP mediaI FSET -- obtain 'media' instruction --
instructions FGET copySelect GETPROP POP copySelectI FSET -- and others --
instructions FGET pageSelect GETPROP POP pageSelectI FSET
instructions FGET onSimplex GETPROP POP onSimplexI FSET
instructions FGET mediaSelect GETPROP POP mediaSelectI FSET
instructions FGET copyName GETPROP POP copyNameI FSET
0 MARK -- any error in the preamble will terminate execution --
iFrame FGET preambleBody FGET
*MAKECOWITHFRAME DOSAVEALL -- execute the preamble --
*LASTFRAME iFrame FSET -- save frame created by preamble --
-- make the required number of copies --
```

```
for copyNumber := 1 to copySelectI FGET *RUNSIZE do {
 -- execute the page bodies --
 copyNameI FGET copyNumber FGET *RUNGET copyName FSET -- set copyName --
 if copySelectI FGET copyNumber FGET *RUNGET then
 for pageNumber := 1 to numPages do {
    0 MARK -- protect against error while executing the page body --
    -- execute the page instructions body --
    0 MAKEVEC pageInstructions FSET -- null instructions in case of mark recovery --
    0 MARK -- separate mark recovery for printing instructions --
        0 MAKEVEC -- null vector is argument --
        iFrame FGET
        pageInstructionsBodies FGET pageNumber FGET GET
        *MAKECOWITHFRAME DOSAVEALL
        while COUNT>1 do { *MERGEPROP }
        pageInstructions FSET
    UNMARK0
    -- decide whether to print this page for this copy --
    if instructions FGET plex GETPROP POP duplex EQ
    pageInstructions FGET pageOnSimplex GETPROP NOT
    { onSimplexI FGET pageNumber FGET *RUNGET } IF OR
    pageSelectI FGET copyNumber FGET *RUNGET pageNumber FGET *RUNGET AND then {
        -- set medium for this page --
        mediaI FGET -- obtain media vector --
            pageInstructions FGET pageMediaSelect GETPROP NOT
            { mediaSelectI FGET copyNumber FGET *RUNGET pageNumber FGET *RUNGET } IF
            GET -- medium description --
        pageNumber FGET -- page number --
        instructions FGET plex GETPROP POP duplex EQ -- true if duplex --
        instructions FGET xImageShift GETPROP POP -- value of xImageShift --
        *SETMEDIUM -- prepare a new image surface --
        -- during execution of the page image body, --
        -- *COPYNUMBERANDNAME will return <copyNumber copyName> --
        iFrame FGET
        pageImageBodies FGET pageNumber FGET GET
        *MAKECOWITHFRAME DOSAVEALL -- call page image body --
    }
    UNMARK0
}}
UNMARK0
```

This program defines the *effect* of executing a master, not necessary 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.

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 § 2).
IMAGE	Any imaging operator (defined in § 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 page instructions bodies may execute only BASE operators. The preamble body may execute only BASE and WEAKIMAGE operators. A page image body may execute any operators. 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 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.

3.2 Environments and names

The fonts, decompressors, and colors which exist outside a master are made accessible through the FINDFONT (§ 4.9), FINDDECOMPRESSOR (§ 4.6.1), and FINDCOLOR (§ 4.7) operators. These, and the *sequenceInsertFile* encoding-notation (§ 2.5.3), are the only links from a master to the outside world. The operation of *sequenceInsertFile* is installation-specific, but it usually names the file to be included with a string which is interpreted as some kind of file name in the installation's file system. The FIND operators use Vectors of Identifiers to name the external objects to be obtained. This naming scheme is open-ended so that a growing set of objects can be named in an orderly and decentralized way.

It is desirable for a preamble to extract the objects needed by its page bodies and save them in the initial frame, since an implementation is likely to handle FGET much more efficiently than FINDFONT, FINDCOLOR, and FINDDECOMPRESSOR.

3.2.1 Identifier names

Identifiers in Interpress provide a general mechanism for naming values which are external to the master, such as fonts supplied by the printer. For external access, identifiers are a necessity in all but the simplest situations. The use of numbers to name fonts, for example, would require a separately maintained list of all the assigned numbers and the font associated with each number. Without this list, a master would not make sense. But maintaining such a list is administratively impractical except in a small community or in one with a single supplier. Identifiers are not only nearly indispensable for font names, they are also quite cheap, since only one external font name needs to appear for each different font used in a master. Hence neither the space occupied by the identifier nor the time required to look it up is a significant consideration.

Identifiers also provide uniformity in naming fonts independently of the conventions of any particular computing system. Identifiers are restricted to letters, digits and "–" characters, and case is not significant; every system should be able to handle such names. Names with structure are represented as Vectors of Identifiers, e.g., *Xerox/TimesRoman/bold* becomes [*Xerox*, *TimesRoman*, *bold*]. This representation makes the structure explicit and removes the temptation to devise ad-hoc conventions for parsing strings to establish structure.

3.2.2 Hierarchical names

It is intended that a *hierarchical* naming system be used for external objects, much like the hierarchical file directory system of many operating systems. This naming convention allows for unlimited growth of the name space without any name conflicts and without the need for any central authority to assign names. For example, all the names defined by Xerox should begin with *Xerox*. Subsequent identifiers in a hierarchical name will further specify the object, e.g., *Xerox/xc82-0-0/TimesRoman*.

Orderly invention of hierarchical names requires the notion of a *registry*, a set of identifiers which controls a particular point in the hierarchical name space. Such points themselves of course have hierarchical names. A registry is an *administrative* entity, maintained by some organization or person with an interest in keeping order in some part of the name space. 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 power to a sub-registry, e.g., for the point *n/m*. In doing so, it resigns its own right to invent names which begin *n/m*, since it cannot be sure that the sub-registry has not already chosen such a name. For example, there might be a registry for *Xerox*, responsible for assigning the names of fonts supplied by Xerox. It might choose to use character-set names at the next level, and invent the names *Ascii*, *Ebcdic*, *xc82-0-0*, etc. The name *Xerox/Ascii/* . . . is quite distinct from *Mergenthaler/Ascii/* . . .

The registry for top-level names (the first elements of hierarchical names) is called the Interpress *universal registry;* identifiers in this registry are called *universal names*. It is intended that an organization wishing to create permanent values which can be referenced reliably from any Interpress master (e.g., ANSI, Xerox, etc.) should obtain an identifier in this registry and thus establish a sub-registry within which to invent names for these objects (see Appendix C).

In Interpress, the term *hierarchical name* refers to a Vector of Identifiers in which the first identifier is a universal name. Hierarchical names are used to name fonts, colors, decompression operators, and printing instructions. It may also be convenient to use universal names at other places in the name space in later versions of Interpress.

3.3 Printing instructions

When a printer is presented with an Interpress master, it must also be given additional information that governs how the master will be printed: the number of copies, what account to charge, an identifying name for the document, etc. These *printing instructions* are not necessarily part of the master itself, because they may be generated separately. 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, will be generated when the demand is made, while 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 in the master.

Printing instructions are represented as a universal property vector in which property names denote particular printing instructions. For example, the vector [*docName*, <Sales report>, *media*, [[*plainPaper*, 0.2159, 0.2794]]] indicates that the printing instruction *docName* is to have the value <Sales report> and the printing instruction *media* has value [[*plainPaper*, 0.2159, 0.2794]]. The standard instructions are described in § 3.3.3.

Because printing instructions are represented in a universal property vector, the instructions vector may contain printer-dependent instructions. For 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 *docOffset*.

3.3.1 Computing the printing instructions

A single printing instructions vector is computed by merging information obtained from several sources.

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, obtained from the protocol used to submit the master for printing, or supplied by the operator of the printer; or it may be empty.

Printing instructions are also obtained by executing the master's *instructionsBody*. This body is executed with the external instructions vector on the stack. After *instructionsBody* is executed, the contents of the stack are merged, using *MERGEPROP, to obtain a single instructions vector. Finally, the printer passes this vector and the external instructions to the *ADDINSTRUCTIONDEFAULTS operator, which may alter the instructions vector in order to enforce printer-dependent defaults, overrides, or operational policies. Interpress does not define all actions of *ADDINSTRUCTIONDEFAULTS, although it does insist on certain default values indicated below.

In normal usage, some instructions will be provided by the external instructions vector and others by the *instructionsBody*. In the simplest case, *instructionsBody* simply pushes a single property vector on the stack, which will be merged with the external instructions vector after the execution of *instructionsBody* is complete. For example, the external instructions will specify the number of copies and the name of the recipient, while the *instructionsBody* will specify the name of the document and the size of paper required.

If *instructionsBody* is null, the external instructions become the instructions because of the way execution of the master is defined.

The *instructionsBody* can control whether its instructions will override external instructions or will function as defaults. Suppose that the execution of *instructionsBody* leaves the stack containing <v_1 *externalInstructions* v_2>, where v_1 and v_2 are property vectors constructed by *instructionsBody* and *externalInstructions* is the original external instructions vector passed as an argument to *instructionsBody*. The act of merging these three property vectors will insure that instructions in v_2 dominate those in *externalInstructions*, which in turn dominate those in v_1.

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 *instructionsBody* is free to compute arbitrary instructions, *ADDINSTRUCTIONDEFAULTS has ultimate control, and may for example reinstate critical external instructions that *instructionsBody* deletes.

3.3.2 The break page

Some of the instructions are used to fill in a *break page* which the printer may provide as a cover sheet for the document being printed and to separate the output from printing of successive masters. The break page gives the document name, creation date, printing date, person who is to receive the output, messages describing errors encountered during printing, etc. Its detailed layout is controlled by the printer.

Information to be printed on the break page and supplied in instructions is contained in vectors of type BreakPageString. A BreakPageString is a Vector of Integers suitable for indexing the font specified by the *breakPageFont* instruction. The main purpose of this font is to indicate the character set of BreakPageStrings. The printer is not required to use this font, but it must print the strings in a font with similar symbols, so that they are readable. Thus if character code 23 corresponds to "A" in this font, whenever 23 is encountered in a 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.

3.3.3 Standard instructions

The following instruction names and meanings are standard. A printer capable of carrying out one of these instructions should interpret the standard instruction as specified here. It is not necessary to specify the value of every instruction; the list below shows *default* values that will be substituted by *ADDINSTRUCTIONDEFAULTS if certain instructions are missing. The description of each instruction gives the name of the instruction (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 is caused. In the text describing each instruction, the symbol *value* is used to denote the value associated with the instruction property.

The parenthesized note in front of the meaning suggests that this instruction is likely to be supplied:

(M)	by the *instructionsBody* in the master;
(E)	in the *external instructions*;
(EM)	in the *external instructions*, with a default value supplied by the master;

This is simply a comment to indicate the intended use of each instruction; there are no restrictions on where an instruction may appear.

breakPageFont	Vector of Identifiers (EM). The name of a font that may be used to print BreakPageStrings. The font will be obtained by <*value* FINDFONT>.
docName	BreakPageString (M). An identifying name for the document.
docCreator	BreakPageString (M). The name of the person who created the document.
docComment	BreakPageString (EM). The string is printed prominently on the break page.
docCreationDate	BreakPageString (M). A string that shows the date and time when the document was created.
docPassword	Vector of Integer (M). If *value*≠<0 MAKEVEC>, the document should not be printed until the operator enters a password such that *AuthenticateFunction*(*password*) = *value*. The definition of *AuthenticateFunction* is printer-dependent. *Default:* <0 MAKEVEC>.

jobSender	BreakPageString (E). The name of the person who requested the document to be printed.
jobRecipient	BreakPageString (E). The name of the person who is to receive the printed output.
jobStartMessage	BreakPageString (E). A message to be displayed to the operator prior to job execution.
jobEndMessage	BreakPageString (E). A message to be displayed to the operator after job execution.
jobStartWait	Integer (E). If *value* is non-zero, the job should not be printed until an operator at the printer instructs the printer to proceed. *Default:* 0.
jobEndWait	Integer (E). If *value* is non-zero, the printer should not print subsequent jobs until an operator at the printer instructs the printer to proceed. *Default:* 0.
jobAccount	Any (E). The value specifies the account to be charged for printing services; the interpretation is printer-dependent.
jobPriority	Identifier (E). The identifier selects one of three printing priorities for this job: *low*, *normal*, or *high*. *Default: normal.*
jobErrorAbort	Identifier (E). This instruction suggests whether a printing job should be aborted when errors occur in the execution of the master. The values may be: *onWarning.* If an error or warning is encountered (§ 5.3), the entire job is aborted. This option is useful if you are printing a final copy of a long job and want to be sure that no errors, such as appearance errors or warnings, will distort the result. *onError.* If an error is encountered (§ 5.3), the entire job is aborted. *default.* The printer will apply a printer-dependent criterion, such as aborting after a certain number of errors of a certain severity have occurred. *struggleOn.* The printer will struggle to print as much as possible of the job, even if many errors are generated. *Default: default.*
jobSummary	Identifier (E). This instruction controls the printing of summary accounting and/or performance information. It can take on four values: *none.* No information is printed. *withDocument.* The summary is printed with the document, e.g., on the break page. *separate.* The summary is printed separately from the document, and may be placed in a separate output bin. *both.* Has the effect of *withDocument* and *separate.* *Default: none.*
jobPassword	Vector of Integer (E). If *value*≠<0 MAKEVEC>, the job should not be printed until the operator enters a password such that *AuthenticateFunction*(*password*) = *value*. *AuthenticateFunction* is defined as for the *jobPassword*

instruction. If both *jobPassword* and *docPassword* instructions are present, two passwords must be supplied. *Default:* ⟨0 MAKEVEC⟩.

finishing Identifier or Vector of Identifiers (EM). This instruction specifies the name of a document-finishing technique to be applied, if any. The identifiers that may appear as values of this instruction are:

none. No finishing is done.

default. A printer-dependent default action is taken.

cornerStaple. A single staple is inserted in each copy of the document 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.

A Vector of Identifiers representing 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: default.

plex Identifier (EM). This instruction tells the printer whether the document is to be printed using only one side of each piece of paper ($value=simplex$) or using both sides of each piece of paper ($value=duplex$). *Default:* either *simplex* or *duplex*, as chosen by the printer.

xImageShift Number (EM). The value of this instruction specifies the distance (in meters) that each page's image is to be shifted in the *x* direction. On odd pages, the shift is to the right if $value>0$ or to the left if $value<0$. On even 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.

media Vector of MediumDescription (EM). This instruction specifies the name and size of each medium that is used to print the master. A MediumDescription is itself a vector, with the following entries:

name: Identifier or Vector of Identifiers (index=0). An Identifier to name the printing medium desired. The name *default* invokes a printer-dependent default medium. Interpress does not define other media identifiers. A universal hierarchical name (§ 3.2.2) may also appear in this element of a MediumDescription.

mediumXSize, mediumYSize: Number, Number (indices=1, 2). The size of the medium in meters. *mediumXSize* and *mediumYSize* are defined in § 4.3.1.

A MediumIndex is an index into the *media* vector that selects a particular medium. The printer may assume that media identified by smaller MediumIndex values are used more frequently than those with larger values. *Default:* [[*default*, 0.2159, 0.2794]].

For example, the instruction [[*default*, 0.2159, 0.2794], [*coverStock*, 0.2159, 0.2794]] establishes that a MediumIndex of 0 identifies default 8½ × 11 inch paper, and a MediumIndex of 1 identifies cover stock, also 8½ × 11 inches. In this example, the cover stock is assumed to be used less frequently than the default paper when printing the document.

The remaining printing instructions deal with selecting 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*) will:

be printed if and only if *copySelect*[*c*]≠0 **and** *pageSelect*[*c.p*]≠0 **and** (printing duplex **or** *onSimplex*[*p*]≠0);

use the medium identified by the MediumIndex equal to *mediaSelect*[*c.p*];

and have 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 page instructions (3.3.4).

The arrays *copySelect*, *pageSelect*, *onSimplex*, *mediaSelect*, and *copyName* referred to are all encoded in the printing instructions. However, in order to keep these arrays compact, a run-length encoding scheme is used. The term "Run of X" refers to a vector that contains alternate values of type Integer and type X. The Integer specifies the number of times the value of X should be repeated in the fully-decoded vector; the fully-decoded vector has a lower bound of 1. It is an 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 is generated 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 Integers [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*: Integer>

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:

```
l,n,j,s: Integer; 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 FGET GET c FSET -- c := r[j+l] --
    if c FGET is not an Integer then error;
    s FGET c FGET ADD s FSET -- s := s+c --
} s FGET
```

<*r*: Vector> <*i*: Integer> *RUNGET → <*value*: Any>

where *value* is the *i*th element of the fully decoded form of the Run of X vector *r*, $1 \leq i \leq$ <*r* *RUNSIZE>. The effect of this operator is defined by the following program:

```
j: Integer; 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 an Integer then error;
    if c FGET ≥ i then goto done;
    i FGET c FGET SUB i FSET; -- i := i−c --
```

Xerox Private Data

```
} error;
done: r j FGET 1 ADD GET -- value := r[j+1] --
```

The following printing instructions are used to define the treatment of all pages and copies:

copySelect	Run of Integer (EM). *Default:* [1, 1], which requests a single copy.
copyName	Run of Identifier (EM). *Default:* [10^7, *null*].
onSimplex	Run of Integer (EM). *Default:* [10^7, 1], which prints all pages when printing simplex.
pageSelect	Run of (Run of Integer) (EM). The outside vector is indexed by copy number, the embedded vector by page number. *Default:* [10^7, [10^7, 1]], which will select all pages for printing on all copies.
mediaSelect	Run of (Run of Integer) (EM). The outside vector is indexed by copy number, the embedded vector by page number. The Integer value is a MediumIndex. *Default:* [10^7, [10^7, 0]], which will select the medium with index 0 in the value of the *media* instruction for all pages and all copies.

The *copySelect* instruction requests that multiple copies be printed. For example, the instruction [*copySelect*, [5, 1]] will print 5 copies. More precisely, it will print 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]] will cause 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]]] will select 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 bodies in the master. Thus [*onSimplex*, [10^7, 1]] will print all pages when printing simplex, even though the document will not have 10^7 pages.

3.3.4 Specifying printing instructions on a page

Certain printing instructions may be specified within a page instructions body rather than computed at the outset. The execution of the page instructions may place on the stack a property vector that contains the following printing instructions:

pageMediaSelect:	The value is a MediumIndex (§ 3.3.3), which selects the medium to use to print this page. This value takes precedence over the value specified by *mediaSelect*[*c,p*] for this page.
pageOnSimplex:	The value is an Integer that is used as the *onSimplex* value for this page. This value takes precedence over the value specified by *onSimplex*[*p*] for this page.

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

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

As an example of the use of page 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, at the outset when the *instructionsBody* is generated, precisely which pages in the master will require which forms. Instead, the creator uses {[*pageMediaSelect*, 0]} or {[*pageMediaSelect*, 1]} as a page instructions body to select an appropriate medium for each page. Of course, the *instructionsBody* must contain an appropriate *media* entry, such as [*media*, [[[*Xerox*, *invoice-first*], 0.2159, 0.2794], [[*Xerox*, *invoice-continuation*], 0.2159, 0.2794]]].

4 Imaging operators

Interpress operators used to 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 on a page 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 page image is a two-dimensional image that accumulates results from primitive images being laid down. It plays the role of the painter's canvas.
- *The mask.* The mask, specified for each primitive image to be added to the page image, 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 color of 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 laid down on the page image. Invocations of mask operators actually cause the page image to be altered. The color to be used is held in a state variable in the imager; it applies to all masks until the color is 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, then, will 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 images are being made on film, the initial state of the film is transparent, which would correspond to white if the film were projected with a white light. Although the discussion of Interpress frequently refers to "pages" for convenience, Interpress imagers can be used to create images on any two-dimensional medium.

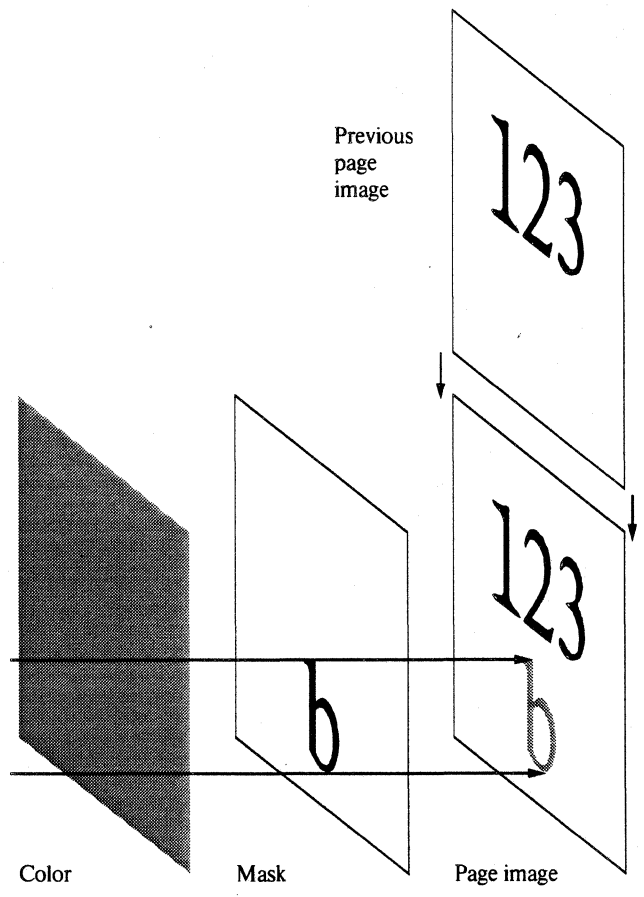


Figure 4.1 The imaging model; application of a mask operator.

4.1.1 Priority

The repeated generation of primitive images opens 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 painted using the same color, reordering the priority of objects will not change the image: objects that overlap will simply appear to merge. Priority order is also unimportant when no two objects on the page image overlap, regardless of the colors used to show the objects. In fact, priority order is important only when objects of different colors overlap.

When priority order is important, the master must inform the imaging operators by setting an 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) the state of 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 Integer index.

⟨*j:* Integer⟩ 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:* Integer⟩ 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 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 reflect the details of the medium selected for printing the page. Specifically, its actions are:

⟨*m:* Vector⟩ ⟨*pageNumber:* Integer⟩ ⟨*duplex:* Integer⟩ ⟨*xImageShift:* Number⟩ *SETMEDIUM → ⟨⟩

where certain imager variables are set:

- Set *mediumXSize* and *mediumYSize* appropriately for the physical medium specified by the medium description *m* (§ 3.3.3). Set *fieldXMin*, *fieldXMax*, *fieldYMin*, and *fieldYMax* as well (§ 4.3.1).
- Set $T := \langle S\ T_{ID}$ 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* will specify a smaller useful x dimension than normally. That is, the x size will be reduced by the absolute value of the image shift.

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	Integer	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
showVec	12	Vector	4.4	[ ]
color	13	Color	4.7	⟨1 MAKEGRAY⟩ (black)
noImage	14	Integer	4.8	0
strokeWidth	15	Number	4.8	0
strokeEnd	16	Integer	4.8	0 (*square*)
underlineStart	17	Number	4.8	0
amplifySpace	18	Number	4.9	1
correctPass	19	Integer	4.10	0
correctShrink	20	Number	4.10	½
correctTX, correctTY	21, 22	Number	4.10	0, 0

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 a pair of (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 to produce 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 to control printing on 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 defined so that the *y* axis points up and the *x* axis to the right, in the normal viewing orientation of the medium 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. While these six variables can be changed by the master using ISET, the imager is insensitive to the changes.

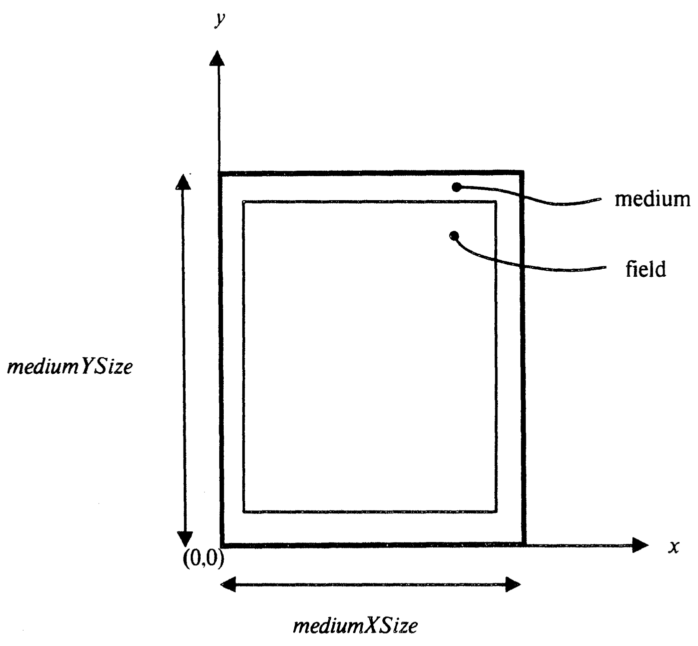


Figure 4.2 Physical medium.

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 may be 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 8½×11 inch paper could have either *mediumXSize*=0.0254×8.5=0.2159 and *mediumYSize*=0.0254×11=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.

For two-sided media, the convention is that to switch from one side to the other, the reader rotates the medium about its y axis.

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 record 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 8½ × 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.

A particularly convenient unit for master coordinates is the *mica*, equal to 10^{-5} meter. For an 8½ × 11 page, coordinates lie in the range $0 \leq x \leq 21590$, $0 \leq y \leq 27940$, which can be represented by a Short Number (§ 2.5.2). The mica offers sufficient precision for most routine typographic needs.

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 linear 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 Integers. 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 will 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 (§ 5.2).

4.3.5 Device coordinate system (DCS)†

The device coordinate system (DCS) is a transformation of the Interpress coordinate system that may be 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 Integers
- There is some ICS point (x, y), $0 \leq x \leq mediumXSize$, $0 \leq y \leq mediumYSize$, 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 § 5.2).

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 measured in one coordinate system into 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, however, 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 multiplied together to form a single *concatenated* 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 to be 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 one:

$$[\, x_{DCS}, y_{DCS}, 1 \,] = [\, x_c, y_c, 1 \,]\, (CT)$$

where C and T have been multiplied together to form a single matrix. By concatenating pairs of transformations, an arbitrary 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.

The printer may restrict the transformations which can be used with character operators and pixel arrays, as described in § 5.1.2.

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 is designed for convenient 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:

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

Note that this operator cannot be called by the master, and is defined only to simplify the definition of the transformation operators which can be called.

⟨*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 will rotate coordinate axes *clockwise* by the angle *a*, while it will rotate 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:

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

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; this is the only way to generate a non-orthogonal transformation.

⟨*m:* Transformation⟩ ⟨*n:* Transformation⟩ CONCAT → ⟨*p:* Transformation⟩

where $p = mn$, i.e., it is the concatenation 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 § 5.2).

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 operators work in conjunction with the current transformation, which is the value of the imager variable *T*, and consequently is saved and restored by DOSAVE and DOSAVEALL operators. 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 operations in TRANS imply 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.

4.4.6 Instancing

Instances of symbols may be placed on the page image by calling composed operators after applying an incremental transformation to *T*. For example, {*m* CONCATT *o* DO} DOSAVESIMPLEBODY will first set the current transformation to ⟨*m T* CONCAT⟩ and then call the operator *o*. Note that because the current transformation is an imager variable, its original value will be restored when DOSAVESIMPLEBODY terminates. Variations on this schema modify the current transformation in other ways before calling the composed operator. A particularly useful form is SHOW:

⟨*v:* Vector⟩ SHOW → -- the effect on the stack depends on the operators called --
: **where** for each *i*, $l \leq i \leq u$, beginning with *v*'s lower bound *l* and ending with *v*'s upper bound *u*, perform {TRANS *showVec* IGET *v i* GET GET DO} DOSAVESIMPLEBODY. In other words, each element of the vector is used as an index into the vector *showVec*, an imager variable. The element extracted from *v* is an operator, which is then called after translating the origin to the current position.

SHOW is useful for generating strings of characters. The value of *showVec* should be a vector of operators, each of which generates a mask for a character in the font. Because each 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 operator in *showVec* 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 → -- the effect on the stack depends on the operators called --

where alternate elements of *v* are taken as indexes into *showVec* 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. The precise effect (if *l* is the lower bound of *v*) is

```
[v l GET] SHOW
v l 1 ADD GET 256 MOD 128 SUB SETXREL
[v l 2 ADD GET] SHOW
v l 3 ADD GET 256 MOD 128 SUB SETXREL
. . .
```

continuing up to the last element of *v*.

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.

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 masks to be defined by *pixel arrays*, arrays of numeric samples of pixels that describe the value of the mask on a two-dimensional grid. This section explains the conventions behind a type named *PixelArray* that is used to represent these arrays.

A PixelArray is constructed with the following primitives:

⟨*xPixels:* Integer⟩ ⟨*yPixels:* Integer⟩ ⟨q_1: Integer⟩ ⟨q_2: Integer⟩ ⟨q_3: Integer⟩
⟨*m:* Transformation⟩ ⟨*samples:* Vector⟩ MAKEPIXELARRAY → ⟨*pa:* PixelArray⟩

where $q_1 = q_2 = q_3 = 1$; other values of the q_i are reserved for future expansion. 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 $0 \leq x \leq xPixels$, $0 \leq y \leq yPixels$. Each pixel is defined by one sample. Each sample value is either 0 or 1. 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 for masks.

A pixel said to be *located* at (p_x, p_y) describes a region of the image centered about the point $(p_x + ½, p_y + ½)$, and extending a distance slightly more than ½ 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.

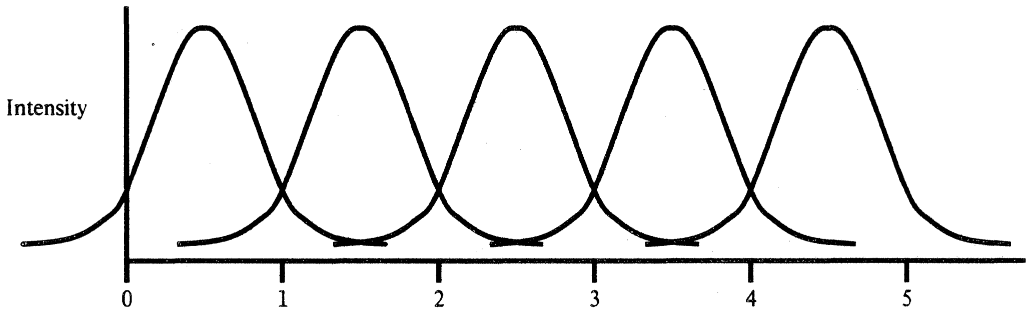
The *samples* vector must contain $xPixels \times yPixels$ samples, each recorded in a separate element in the vector. The sequence of samples in the vector is such that a rectangular grid is scanned out in a series of scan-lines. The first pixel in the vector is located at $(0, 0)$, the next pixel at $(0, 1)$, and so forth up to $(0, yPixels - 1)$: thus the first *yPixels* elements of the *samples* vector determine a 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 $(xPixels - 1, yPixels - 1)$.

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 as a mask (§ 4.8.4) or to define 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 obtained from *T* by concatenation. For a mask, *um* is *T* when MASKPIXEL is executed; for a sampled color, it is the *um* argument to MAKESAMPLEDBLACK. The *net* transformation for the pixel array <*xPixels yPixels* 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. The imager may specify a set of *easy* values of *nm* which it handles efficiently, and it may be unable to handle a pixel array at all unless *nm* is easy (§ 5.1.2).



Distance (coordinate in standard coordinate system)
Example shows an image 5 pixels wide, i.e., xPixels=5.

Figure 4.3 Pixel Intensity profiles.

4.6.1 Compressing sample vectors

When a large pixel array is included in an Interpress master, its *samples* vector must usually be compressed in some way. The compressed data is expressed as a single vector, together with an operator that can be called to "decompress" the data in the vector into the expanded form of the *samples* vector. A decompression operator is called as though it were a primitive operator with the definition:

<*v:* Vector> *decompress* → <*samples:* Vector>

> **where** *v* contains the compressed pixel data and any additional parameters the *decompress* operator may need.

Note: decompress is not in fact a primitive operator supplied by the imager. It is used here only to illustrate the form of a decompression operator. In practice, the operator will be called with some form of DO.

A decompression operator is obtained by the FINDDECOMPRESSOR primitive:

<*v:* Vector> FINDDECOMPRESSOR → <*o:* Operator>

> **where** *v* is a Vector of Identifiers which is the hierarchical 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.

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 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 black-and-white sampled on a raster. A value of type *Color* fully specifies a color.

A *constant* color deposits the same intensity of ink 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:* Color>

> **where** *col* represents a shade of gray specified by *f*, the fraction of incident light energy absorbed by the ink. Thus $f=0$ will yield the paper color and $f=1$ will print black.

<*v:* Vector> FINDCOLOR → <*col:* Color>

> **where** *v* is a Vector of Identifiers which is the hierarchical name of the desired color. If the specified color cannot be found, an approximation to it is returned. Examples of color names might be *Xerox/highlight* or *nbs/cns/bluegreen*.

A *sampled black-and-white* color allows the presentation of stipple patterns, photographs, and other images with rapidly varying colors. The idea is to specify the color (black or white/transparent) 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> <*transparent:* Integer> MAKESAMPLEDBLACK
→ <*col:* Color>

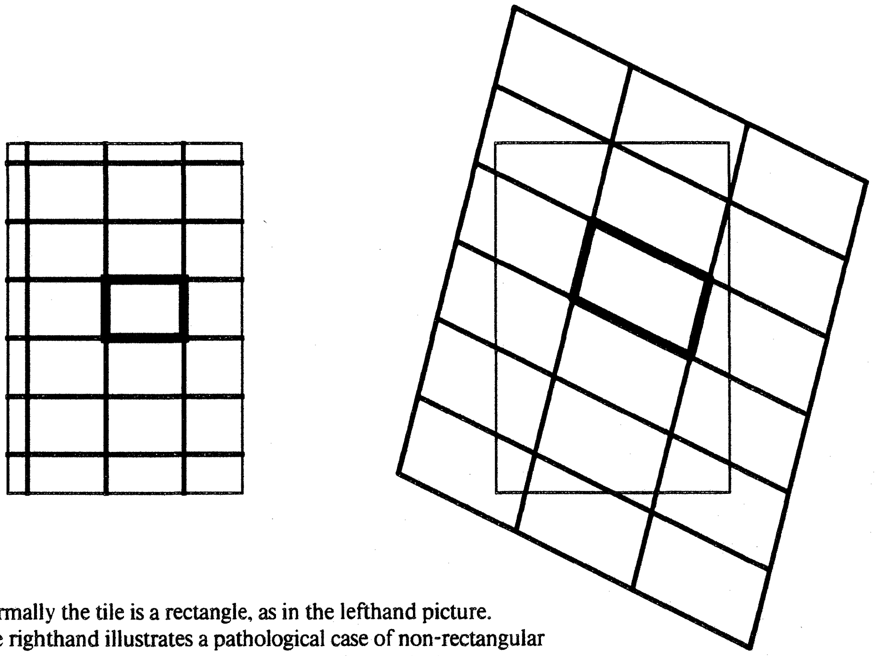
> **where** *pa* provides color samples, *um* is a transformation that maps the region defined by *pa* to device coordinates, and *transparent* is 0 or 1.

A sampled black-and-white color specification contains three components: a PixelArray *pa* containing the samples, a transformation *um*, and a *transparent* parameter which is 0 or 1. If *pa* was made by <*xPixels yPixels* 1 1 1 *pm samples* MAKEPIXELARRAY>, then <*pa um transparent* MAKESAMPLEDBLACK> 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 deposited through the mask on an image pixel corresponding in position to a color 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 *transparent:* for *transparent*$=0$ the color is white, i.e., ⟨0 MAKEGRAY⟩; for *transparent*$=1$ no ink is deposited, i.e., it is as if the mask did not cover this pixel.

Note that the use of *transparent* is an exception to the general imaging model explained in § 4.1. Generally, colors are opaque, and, when applied, obliterate any previous color at the same point. By contrast, those portions of a transparent sampled black-and-white color with a sample value of zero will not obliterate or alter any previously-applied color.

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 MAKESAMPLEDBLACK 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 lefthand picture. The righthand 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.

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 is also a convenience operator for setting it to a constant gray.

⟨*f*: Number⟩ SETGRAY → ⟨⟩
 where the effect is *f* MAKEGRAY *color* ISET.

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 sets of pre-defined operators called *fonts* (§ 4.9). Mask operators are also available to make images of rectangles, line drawings, or filled polygons, 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 variables are:

- *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*≠0 is preserved (§ 4.1.1).
- *noImage*. If *noImage* is non-zero, any operator with MASK in its name 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.

4.8.1 Geometry: trajectories and outlines

Shapes are defined geometrically in terms of *segments*, *trajectories* and *outlines*. A *segment* is a directed line 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 as a mask. At this point, the numbers describing the trajectory or outline are interpreted as defining geometry in the master coordinate system, which the imager operators MASKFILL and MASKSTROKE convert 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, placing on the stack a *trajectory* value describing the trajectory. Then the trajectory is extended by LINETO, which adds 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_p LINETO⟩, where (x_p, y_p) is t_1's *lp*.

⟨t_1: Trajectory⟩ ⟨*y:* Number⟩ LINETOY → ⟨t_2: Trajectory⟩
  where the effect is ⟨t_1 x_p y LINETO⟩, where (x_p, y_p) is t_1's *lp*.

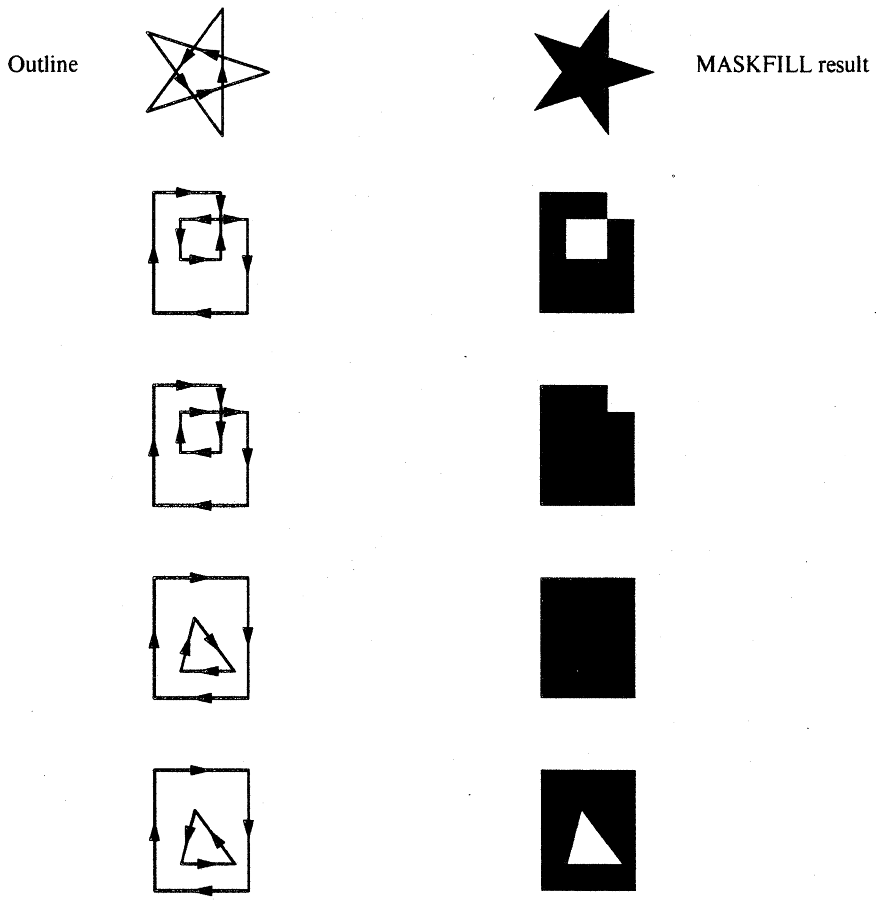


Figure 4.5 Examples of winding number conventions.

An *outline* is represented by a separate type, built using the operator:

⟨t_1: Trajectory⟩ ⟨t_2: Trajectory⟩ . . . ⟨t_n: Trajectory⟩ ⟨*n:* Integer⟩ MAKEOUTLINE
  → ⟨*o:* Outline⟩
  where the trajectories t_1, t_2, . . . t_n together form an outline. Each of the trajectories will be closed if necessary.

The MASKFILL primitive that takes an outline as its argument needs to decide which points lie "inside" the outline. If none of the trajectories in the outline intersects itself or another trajectory, the inside of the trajectory is unambiguous. In other cases, 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: it is the number of closed trajectories in the outline that are wound anti-clockwise around the point, minus the number of closed trajectories in the outline that are wound clockwise around the point. Interpress uses the convention that points with non-zero winding number lie inside the outline. Figure 4.5 illustrates several outlines and shows their "insides" according to this convention. Note that for multi-trajectory outlines, the order in which points on a trajectory are specified is important.

A creator may prefer to use another convention for determining the inside of an outline, but must convert this convention to the Interpress one when making a master. Two other popular conventions are "odd winding number is inside" and "positive winding number is inside."

4.8.2 Filled outlines and strokes

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.*

Another operator creates a mask from a trajectory, which it uses to define the center-line of a stroke to be drawn on the page image:

⟨*t:* Trajectory⟩ MASKSTROKE → ⟨⟩

where the trajectory *t* is first broadened to have uniform width specified by the imager variable *strokeWidth,* fitted with the endpoints specified by the imager variable *strokeEnd,* then transformed into device coordinates by the current transformation *T,* and used as a mask to alter the page image.

The stroke is created from the trajectory by building a stroke of width *strokeWidth,* an imager variable, using the trajectory as the center-line. Joints between segments of the trajectory are *mitered,* i.e., sides of the stroke are extended until they meet. Segments of a trajectory meeting at an acute angle will thus generate long, sharp corners.

The treatment of the two ends of the trajectory is controlled by the value of *strokeEnd,* another imager variable. Figure 4.6 illustrates the three options:

strokeEnd=0 (*square*). A butt end is formed after extending the line 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 at the specified endpoint.

strokeEnd=2 (*round*). Ends are capped with a semicircle whose diameter is the same as the line width and whose center coincides with the trajectory endpoint.

If *butt* or *square* end geometry is undetermined because the trajectory starts or ends with a segment whose start and end points coincide, an appearance error is generated instead of a mask.

To generate strokes with rounded joints between segments, rather than mitered joints, MASKSTROKE should be called separately for each segment, using *strokeEnd*=2 (*round*).

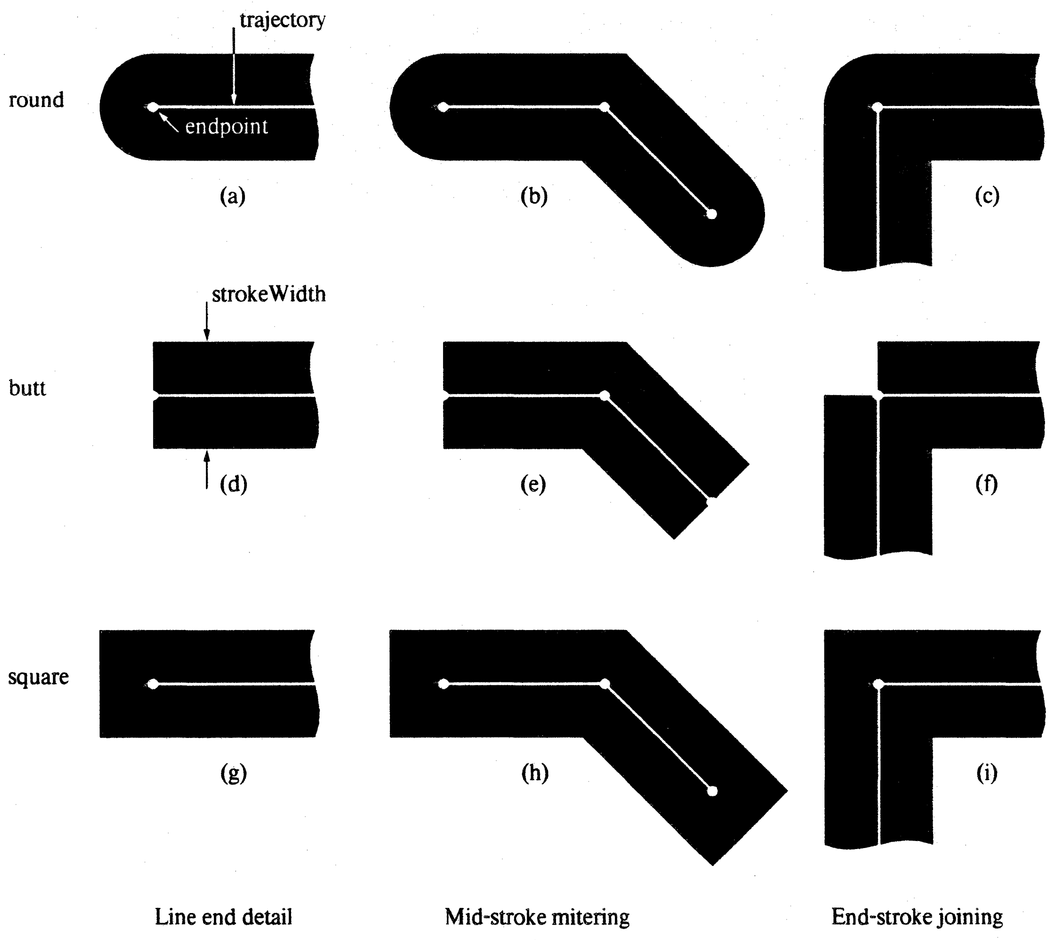


Figure 4.6 Line types for MASKSTROKE.

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.

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
} DOSAVESIMPLEBODY,
```

i.e., a rectangle of width *w* and height *h* is drawn 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.

Xerox Private Data

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 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 or current position --
```

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 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.

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.

4.9 Character operators

It is possible to make an image of any character using the mask commands already introduced: pixel arrays or filled polygons 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 operator 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.

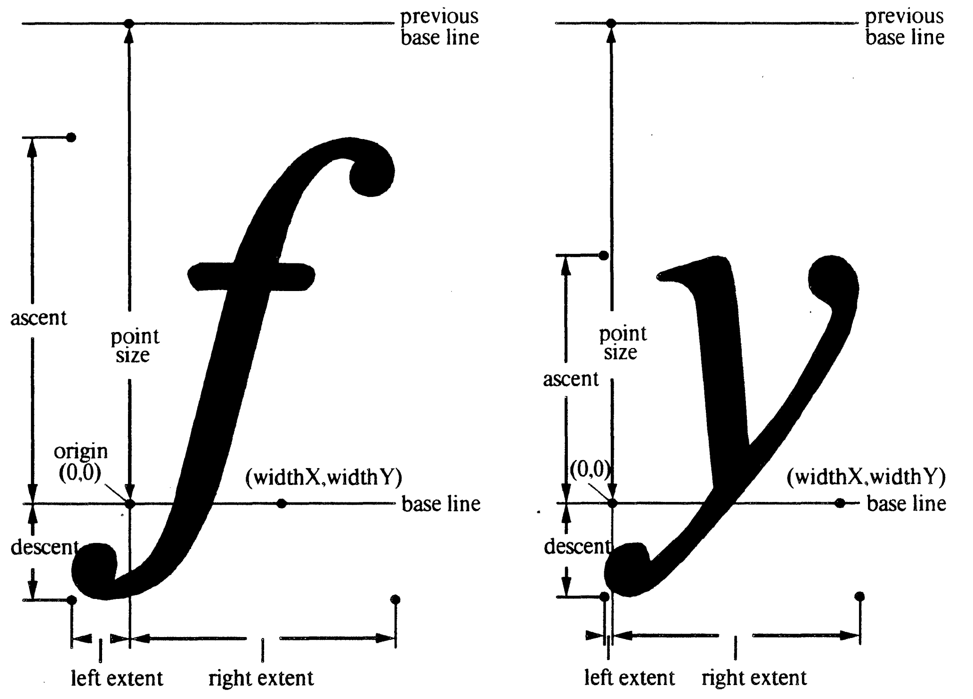
Each character is represented by a composed operator called a *character operator.* Instances of characters are then placed on the page by invoking these character operators with suitable transformations. In order for the creator to anticipate the effect of these invocations, all these operators observe common conventions. This section defines these conventions.

A character operator performs three operations:

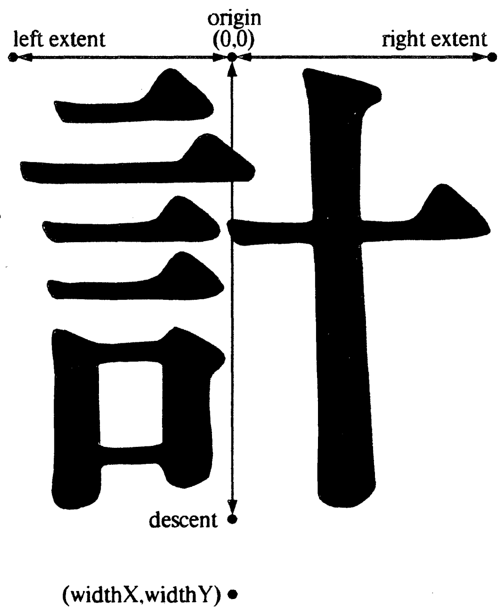
1. *Generates masks.* It invokes mask operators to specify the mask or masks that define the shape of the character, thus causing an image of the character to be added to the page image. The placement, size and orientation of the mask are controlled by the current transformation.
2. *Moves to next character 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.
3. *Corrects spacing.* Small adjustments to the current position may be made to compensate for inaccuracies in character widths.

Generally, character operators have the simple form outlined above. In principle, however, a character operator may be an arbitrary composed operator that executes arbitrary computations; such operators are still subject to the constraint that each operator must alter the current position in a standard way (see below). The detailed design of character operators is not dictated by Interpress, but is left to font and character-set designers.

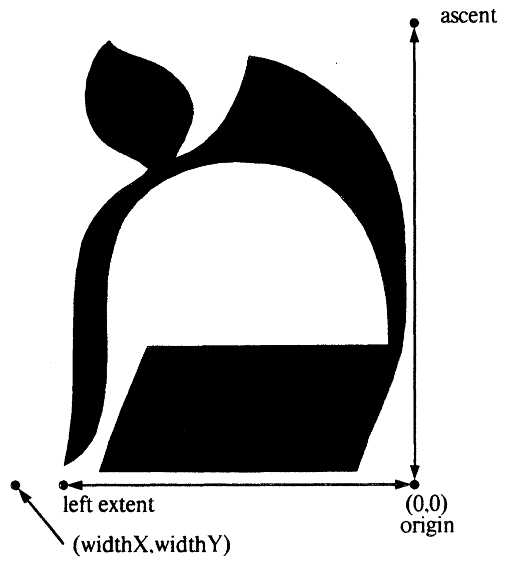
Generating masks. Figure 4.7 shows examples of the masks defined by various character operators and the various measurements that are made on them. A character is defined in the *character coordinate system.* The masks have an *origin,* shown as (0, 0) in the figures. If the character operator is invoked with SHOW, 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.



(a) Metrics for left-to-right Roman characters

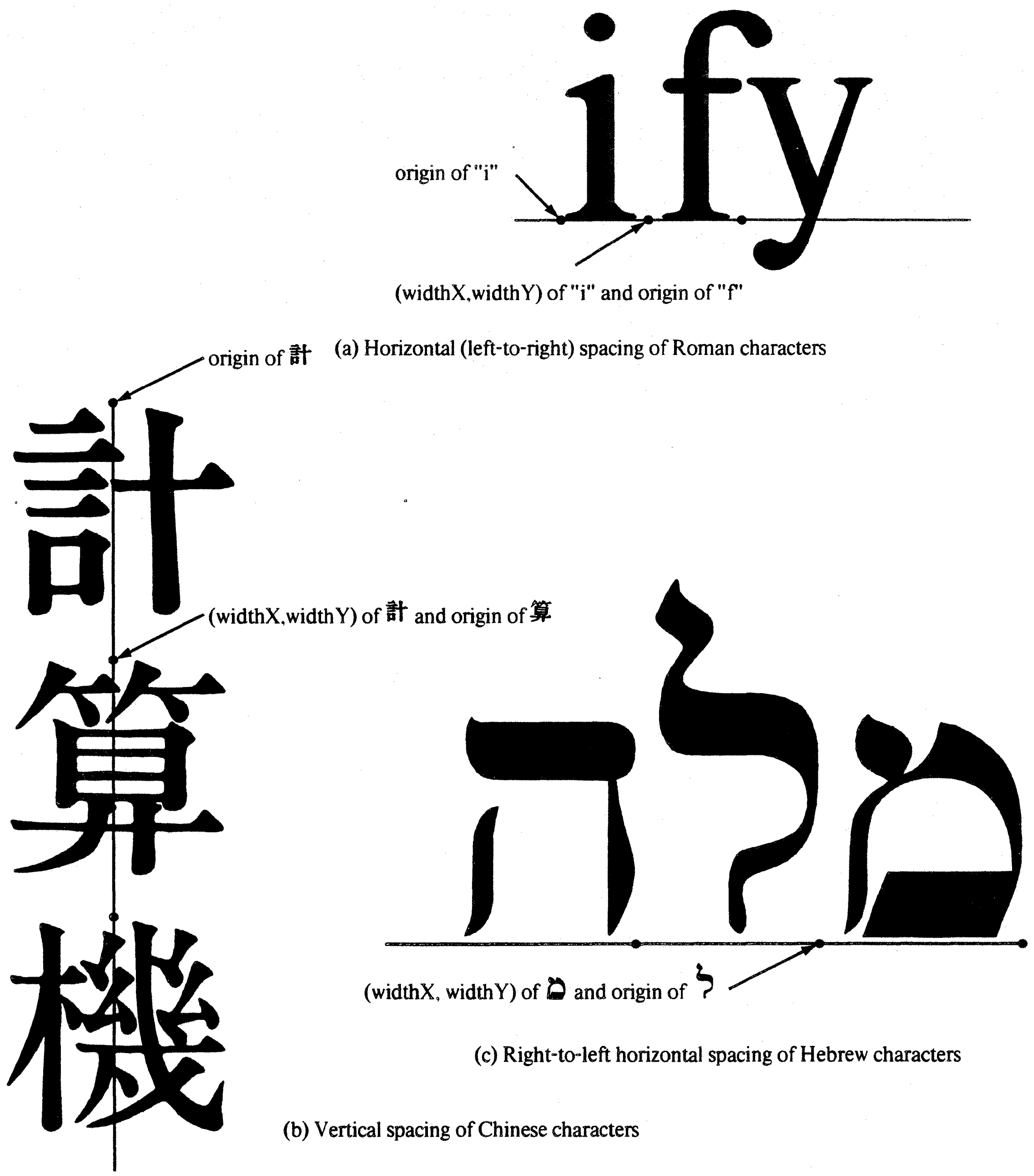


(b) Metrics for top-to-bottom vertical spacing



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

Figure 4.7 Character metrics.


origin of "i"
(widthX,widthY) of "i" and origin of "f"
origin of 計
(widthX,widthY) of 計 and origin of 算
(widthX, widthY) of מ and origin of ל

(a) Horizontal (left-to-right) spacing of Roman characters

(b) Vertical spacing of Chinese characters

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

Figure 4.8 Spacing character masks.

In Latin alphabets, such as the italic font shown in Figure 4.7a, 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 4.7b has its origin at the top center; it is intended to be used for setting characters vertically top-to-bottom. The Hebrew character in Figure 4.7c has its origin at the lower right; it is intended to be used for setting characters horizontally right-to-left.

The units of measurement are defined to correspond to the "size" of the character, as specified in the printing industry: *a distance of 1 unit is defined as the "point size" (or "body size") of the character.* This convention is illustrated in Figure 4.7a: the point (0, 1) is the origin of a character on a line above this one, spaced above this one by the "point size" of the character. The actual size of the character that will be placed in the image is controlled by the transformation that is current when the character operator is invoked.

Moving to the next character position. After the mask is defined by the character operator, the current position is altered by executing *widthX widthY* SETXYREL. The parameters *widthX* and *widthY* are part of the character operator definition. Informally, they determine the "width" of the character mask just imaged, as shown in Figure 4.7. More precisely, they determine where the origin of the next character should (usually) be placed.

For most western languages, characters are read horizontally, so *widthY* will normally be zero and *widthX* will be positive. This will yield left-to-right spacing as shown in Figure 4.8a. Traditional Chinese characters, which are set top-to-bottom, might have a zero *widthX* and a negative *widthY* to establish a current position for a character immediately below the present one; this is illustrated in Figure 4.8b. Masks intended to represent Hebrew characters, which are set right-to-left, may have negative *widthX* values, as shown in Figure 4.8c.

When text is justified between fixed margins, the width of "spaceband" characters is adjusted so that the words on the line appear evenly spaced. A character operator can achieve the effect of a spaceband by using a slightly different spacing computation, namely *widthX*amplifySpace widthY*amplifySpace* SETXYREL, where *amplifySpace* is an imager variable. Characters using this convention are termed *amplifying* characters, as their width is determined in part by the font designer, who specifies *widthX* and *widthY*, 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 widths (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 *widthX widthY* CORRECTSPACE if the character is not amplifying, or *widthX*amplifySpace widthY*amplifySpace* CORRECTSPACE if it is. If the character's width should not be adjusted (e.g., a character which deposits ink, or a "figure space" designed to equal precisely the widths of the figures 0..9), the operator calls CORRECTMASK.

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 operator for an acute accent, with $widthX = widthY = 0$. 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 Fonts

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

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

operators: Vector of composed character operators. This vector is called a *font.* An individual character operator is thus identified by a font and a *character index* within the font vector. Interpress establishes no conventions for the correspondence between character shapes and character indices. In this way, a font may represent an arbitrary "character set," i.e., an arbitrary mapping from character indices to shapes.

characterMetrics: Property vector of CharacterMetrics. Corresponding to each character operator in *operators* is a name, value pair in *characterMetrics* in which the property name is the character index and the property value is a CharacterMetrics vector, metric information for each character (§ 4.9.3).

metrics: Property vector. This Metrics vector contains important metric information about the font as a whole (§ 4.9.3).

name: Vector of Identifiers. The hierarchical name of the font.

Metric information is described more fully in § 4.9.3. Only the *operators* vector of a font stored in the printer is accessible to the master.

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/xc82-0-0/TimesRoman.* the identifier *xc82-0-0* 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.
- *Viewing 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, or "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 "viewing size" as three discrete values (*footnote*, *body*, *headline*) or in a more continuous way (*ViewingSize−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 hierarchical name. So an identifier like *version102* might be appended to the example above to indicate the version.

The *name* element of a FontDescription contains its full hierarchical name, and thus serves to identify the font unambiguously. When a master presents a font name to be looked up in the printer's font library, that font should be supplied if the printer has it. Otherwise, the closest available approximation should be supplied. The FINDFONT operator, which looks up fonts, does the best it can to find a suitable font, giving an appearance error if it has to approximate.

⟨*v:* Vector⟩ FINDFONT → ⟨*w:* Vector⟩

where *v* is a Vector of Identifiers, which is the hierarchical name of the font. The result *w* is the *operators* element of the best approximation to this font which the printer can find in its library.

4.9.2 Modifying a character vector

For imaging purposes, the master usually wants to modify the operators of a font in certain ways. The MODIFYFONT operator applies a transformation to each operator in the *operators* vector, intended to allow all characters to be scaled to a particular size.

<*v:* Vector> <*m:* Transformation> MODIFYFONT → <*w:* Vector>

> **where** *v* is a vector of operators, usually a result of FINDFONT. The result *w* has the same SHAPE as *v*, and is obtained from *v* by replacing each element *e* by {*m* CONCATT *e* DO} MAKESIMPLECO.

Usually the master saves the vector *w* in a frame for future reference. The SETFONT operator sets the *showVec* imager variable from an element of the frame. SHOW (§ 4.4.6) can then be used to image characters using this font.

<*n:* Integer> SETFONT → <>

> **where** the effect is FGET *showVec* ISET; i.e., the current font (*showVec*) is set to the *n*th element of the current frame.

The *net* transformation applied to a font operator when it is 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 supplied to MODIFYFONT; *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 (§ 5.1.2).

4.9.3 Metrics

Because the creator must make numerous formatting decisions when creating a master, it needs to know various metric information about font characters in order to build an Interpress master that will produce the desired image. It needs to know the width of each character and whether the width is scaled by *amplifySpace*, since incremental positioning computations are made by the imager (i.e., by altering the current position within a character operator) that must be correctly anticipated by the creator. The creator may also need to know how to place subscripts, superscripts, or accents with respect to a particular character. It may wish to *kern* character pairs by changing the spacing between them slightly. For formatting mathematical expressions, it may wish to choose sizes of brackets or parentheses that match the size of a built-up expression.

This metric information is communicated to creators through an Interpress master whose preamble, when executed, leaves on the stack one or more of the FontDescription vectors described in § 4.9.1; the *operators* property need not be specified. The only literals in this master are Numbers, Identifiers, and the primitive operators MAKEVEC, MAKEVECLU, ROTATE, SCALE, SCALE2, and CONCAT. No encoding notations are used.

The *characterMetrics* element of a FontDescription is a property vector of character metric information. Corresponding to each character index *i* that represents a character in the font whose FontDescription is *fd* is a CharacterMetrics vector obtained by <*fd characterMetrics* GETPROP POP *i* GETPROP POP>. A CharacterMetrics vector is a property vector of metric information about a single character. The property names and corresponding meanings of elements are given below. All dimensions are recorded in the standard character coordinate system described in Figure 4.7. While it is not necessary that all properties of a character be recorded, the *widthX*, *widthY*, and *amplified* properties must be present for each character (unless the default values given for these properties are correct).

widthX: Number.
If the *widthX* property is not present in a character metrics vector, the value of *widthX* may be assumed to be zero.

widthY: Number.
If the *widthY* property is not present in a character metrics vector, the value of *widthY* may be assumed to be zero.

amplified: Integer.
1 if the character operator executes *widthX*amplifySpace widthY*amplifySpace* SETXYREL, and 0 if it executes *widthX widthY* SETXYREL. If the *amplified* property is not present in a character metrics vector, the value of *amplified* may be assumed to be zero.

correction: Integer.
Indicates what operator, if any, is called to correct spacing: 0=none, 1=CORRECTSPACE, 2=CORRECTMASK.

leftExtent, rightExtent, descent, ascent: Number
These four numbers describe the *bounding box* of the character mask, as measured from the character origin. The numbers are all signed, and represent distances to the left, right, bottom and top respectively; see Figure 4.7a. For example, the *descent* of a single quote character (') will usually be negative. All four numbers are zero if the character contains no mask (i.e., if it is a space).

centerX, centerY: Number.
The coordinates of the *optical center* of the character.

kerns: Vector.
This value suggests spacing adjustments to use, based upon various *successor* characters in the same font. Each element of the *kerns* vector is itself a three-element Vector that specifies:

successor (index 0): Integer. The index of a successor character.

kernX (index 1), *kernY* (index 2): Number.
The amount to add to the character's width (in the character coordinate system) to kern this character to the successor character.

ligatures: Vector.
This value suggests ligature substitutions based upon various successor characters in the same font. Each element of the *ligatures* vector is itself a two-element Vector that specifies:

successor (index 0): Integer. The index of a successor character.

ligatureCharacter (index 1): Integer. The index of a ligature character to use in place of the two-character sequence comprising this character and the successor character. Note that the ligature character can also have a ligature, thus allowing arbitrarily long ligature sequences to be specified.

superscriptX, superscriptY, subscriptX, subscriptY: Number.
These values suggest locations relative to the character's origin at which to place superscripts and/or subscripts.

The *metrics* element of a font is a property vector. The names and meanings of the elements are given below:

easy: Vector of Transformations.
This vector describes the sizes and rotations of the font that the imager can handle easily. Although Interpress allows character operators to be called with arbitrary transformations, device limitations may prevent precise rendition of characters in all cases (see § 4.9.4). The *easy* vector suggests character sizes and orientations that will be printed with greatest fidelity. Each element in the vector is a transformation from the character coordinate system to the ICS that describes a combination of rotation and scaling that the printer can accommodate easily; for example, the transformation <352778/100000000 SCALE> describes a 10-point size oriented to point up in the normal viewing orientation. A transformation <0 SCALE> indicates that the font may be used easily with an arbitrary transformation.

xHeight: Number.
Pertains only to Latin alphabets. The height of lower-case characters in the font.

slant: Number.
Given in degrees, this is the slant (to the right of the vertical) of the characters defined by the character operators. This value can be used to position accent marks. For example, the slant of a Times Roman font would be 0 degrees; the slant of a Times Italic font might be 7 degrees.

underlineOffset: Number.
Distance below baseline where the top of an underline bar should appear.

underlineThickness: Number.
Thickness of desirable underline bar.

Creators may wish to retain additional information about fonts, such as the maximum bounding box of all characters in the font, or the maximum width, or whether all characters in the font have the same widths. This information need not be represented explicitly in the font-wide metric information, as it can be derived by inspecting all the character metrics for the font.

4.9.4 Fallback positions for characters

The chief difficulty that an imager may encounter when printing characters is 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 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 := correctMaskCount1; --*--
    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 := $T_v(x, y, T)$
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*$(0, 0, dx, dy)$.

Xerox Private Data

The *COMPUTECORRECTIONS calculation itself is:

```
$correctMaskX := 0$; $correctMaskY := 0$; --*--
$correctMaskCount := correctMaskCount - 1$; --*--
$correctSpaceX := correctTargetX - DCScpx$; --*--
$correctSpaceY := correctTargetY - DCScpy$; --*--
-- Compute ratio of space correction to available space --
$f := length(correctSpaceX, correctSpaceY) / length(correctSumX, correctSumY)$ --*--
-- Test if line too long and space-correction threshold is exceeded --
if ($distance(correctcpx, correctcpy, correctTargetX, correctTargetY) <$ --*--
 $distance(correctcpx, correctcpy, DCScpx, DCScpy)$ and $f > correctShrink$) or --*--
 $length(correctSumX, correctSumY) = 0$ then begin --*--
    -- Must reposition masks too --
    $correctMaskX := correctSpaceX + correctShrink * correctSumX$; --*--
    $correctMaskY := correctSpaceY + correctShrink * correctSumY$; --*--
    $correctSpaceX := correctSpaceX - correctMaskX$; --*--
    $correctSpaceY := correctSpaceY - correctMaskY$; --*--
    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 widths assumed by the creator. However, in the cases where creator and printer are in exact agreement about widths, 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":

$\langle x$: Number$\rangle$ $\langle y$: Number$\rangle$ SETCORRECTMEASURE → $\langle\rangle$
 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	Integer	
correctShrink	Number	Allowable space shrink
correctTX, correctTY	Number	Line tolerance
CORRECT **variables (persistent)**, not directly available to the master:		
correctMaskCount	Integer	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 important practical issues connected with the implementation and operation of creators and imagers.

5.1 Subsets

Interpress provides an extensive set of facilities for describing images. Many printers are not able to produce all the images which can be specified in Interpress. There are three factors which determine the set of images a printer can produce:

- Its *subset:* roughly, the set of types and primitives that it supports. The standard subsets of Interpress are defined in § 5.1.1, and informally tabulated in Table 5.1. A *subset S printer* supports at least all the facilities in subset *S*. The subsets provide a rough characterization of the power of a printer, as well as some guidance to the designer about what facilities it is desirable to include or omit as a whole.
- Its *environment:* the fonts and decompression operations that it makes available to a master through the FINDFONT, FINDCOLOR, and FINDDECOMPRESSOR primitives, and the list of *easy* net transformations (§ 5.1.2). Each printer must specify what fonts, colors, decompressors and easy transformations it offers. Interpress does not define how this is done. (But see § 4.9.3.)
- The *complexity* of the images it can handle. This difficult question is discussed in § 5.1.3.

§ 5.1.1 defines the minimum set of facilities that every Interpress printer will support; these facilities constitute *text Interpress.* It also specifies a small number of *enhancement modules*, which are groups of optional facilities. A *subset* of Interpress is defined by choosing one module from each row of Table 5.1. A module is defined by listing the types, primitive operators and literals included, and stating any restrictions on how the operators may be used.

A subset is named by a property vector which specifies the module chosen from each row of Table 5.1. A *text* module is named by the identifer *Text.* Enhancement modules are named by the identifiers indicated in the table. For example, if polygons and gray are included in addition to text Interpress, the *subset name* is [Language, *Text*, Graphics, *Polygons*, PixelArrays, *Text*, Ink, *Gray*, Limits, *Text*]. Informally, a subset can be named by specifying the increments to text Interpress, e.g., *Text* with *Polygons* and *Gray.*

Table 5.1 Informal Interpress subset definitions

	Text	*Enhancement*
Language	No control, test or arithmetic primitives	*Computation:* IF etc., EQ etc., ADD etc.
Graphics	Horizontal and vertical strokes, translation, scaling, 90° rotation	*Polygons:* all transformations, all straight lines, filled outlines
PixelArrays	No	*Binary*
Ink	Black	*Gray*
Limits	As in Table 5.2	*Full:* no individual limits

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

5.1.1 Subsets and modules

Text Interpress consists of the following facilities:

- All the types and literals of the base language, and all the operators of §§ 2.4.3-6 (vectors, frames, operators and the stack). Nothing from §§ 2.4.7-9 (control, test and arithmetic).
- All the facilities of Chapter 3.
- All the types and operators of § 4.2 (imager variables), § 4.4 (transformations), and § 4.5 (current position). Nothing from § 4.6 (pixel arrays) or § 4.7 (color). Arguments to ROTATE must be integer multiples of 90.
- From § 4.8 (masks) only MASKRECTANGLE, STARTUNDERLINE, MASKUNDERLINE, and MASKVECTOR. The Outline and Trajectory types are excluded. When MASKVECTOR is executed, *strokeEnd* must be 0 or 1 (no rounded ends), and either $x_1=x_2$ or $y_1=y_2$ (horizontal or vertical strokes only).
- All the operators of § 4.9 (characters) and § 4.10 (correction).
- The minimum limits specified in Table 5.2.

Note that "text Interpress" provides, in addition to text imaging operators, the ability to make horizontal and vertical strokes and rectangles. Moreover, text may be oriented horizontally or vertically, subject to the easy net transformations available (§ 5.1.2).

Table 5.2 Minimum values for size limits

Name	Where defined	Minimum limit
maxInteger	(§ 2.2.1)	$2^{24}-1$
maxIdLength	(§ 2.2.2)	100 characters
maxBodyLength	(§ 2.2.5)	10000 literals
maxStackLength	(§ 2.3.1)	1000 values
maxVecSize	(§ 2.2.4)	1000 elements
topFrameSize	(§ 3.1)	50 elements

The enhancement modules are defined as follows. Each module also includes all the contents of modules to the left of it in Table 5.1.

Language

Computation: All the operators of §§ 2.4.7-9 (control, test and arithmetic).

Graphics

Polygons: The Trajectory and Outline types, and MOVETO, LINETO, LINETOX, LINETOY, MAKEOUTLINE, MASKFILL, MASKSTROKE, MASKTRAPEZOIDX, MASKTRAPEZOIDY from § 4.8. Any arguments to ROTATE and MASKVECTOR. Any defined value for *strokeEnd.*

Pixel arrays

Binary: The PixelArray type, and MAKEPIXELARRAY and FINDDECOMPRESSOR from § 4.6, and MASKPIXEL from § 4.8.

Ink

Gray: MAKEGRAY, SETGRAY, FINDCOLOR, and MAKESAMPLEDBLACK from § 4.7, subject to the restriction that MASKPIXEL cannot be called when a sampled color is in effect.

Limits

Full: The sizes of objects other than *maxInteger* are limited only by the total capacity of the interpreter and imager; more restrictive limits on individual objects or classes of objects are not enforced.

5.1.2 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 ICS that the printer can handle efficiently; these are the *easy* net transformations, which contain only scaling and rotation components. A printer is assumed to be able to handle arbitrary translations. A printer may refuse to handle a sampled color, PixelArray mask, or font operator which has a net transformation not in the easy set. For a font, the easy transformations are specified in the *easy* element of the metrics vector (§ 4.9.3). Interpress does not define how easy pixel array transformations are specified.

5.1.3 Image complexity

It would be nice if a subset *S* printer (i.e., one that supports all the features of subset *S* Interpress) could guarantee to print any subset *S* master. A printer that can do this is called *unlimited;* other printers are *limited.* It is highly recommended that printers should be unlimited, even if performance is greatly degraded on complex images. The reason is that most pages of a master are typically of uniform complexity, but only one very complex page is needed to render the entire master unprintable.

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, or else the image is spoiled. An asynchronous device has no such requirement, and can normally be used in an unlimited printer without further ado. To make an unlimited printer with a synchronous device, 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 limited printer, one must have some characterization of the complexity of the images it can handle. 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. The lack of decent lower bounds, however, has serious implications for clients, since if only one page in a 50 page document cannot be printed on a particular printer, that printer is much less useful (of course it will still print the master, with an appearance error for the troublesome page).

5.1.4 Performance

Not all masters are equally easy to execute because different Interpress operators and constructs entail different amounts of computing. Moreover, certain printers may be able to process certain masters quickly, while slowing down for others. Documentation for a printer may specify the properties that a master should have to be executed efficiently. A printer of a particular subset is required to support all facilities defined for that subset in § 5.1.1, but is free to support only some of these facilities efficiently.

5.2 Numeric precision†

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

Xerox Private Data

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 $s_{max}/s_{min}<16$, $s_{ma}<10^{20}$, 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.

5.3 Error handling

The effect of an 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. For example, if the entire image is reduced in size compared to the size specified, an appearance warning is issued.

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. For example, if an imager cannot display a filled mask that is represented in the master, an appearance error is generated.

Master warning:

These errors mean that something is amiss in the specification of the master, but the error is not severe. For example, arithmetic overflow will cause a master warning.

Master error:

These errors signal severe problems in interpreting the master. It may be necessary to abandon further interpretation of the master and to simply print a page that describes the error.

Appendix A References

American National Standards Institute. *American National Standard Code for Information Interchange.* ANSI X3.4-1977.

United States version of ISO 646.

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 646–1973 (E).

This document defines a limited character set for information interchange. It is almost compatible with ASCII. (next reference). 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. *Interpress 82 Reader's Guide.* Xerox System Integration Guide. Stamford, Connecticut; 1982 May; XSIG 018205.

An overview of Interpress, including a paragraph-by-paragraph commentary on portions of the standard. XSIG 018205 corresponds to an earlier version of the standard; most of its contents, however, apply to Interpress 2.0 as well.

Xerox Corporation. *Introduction to Interpress.* Xerox System Integration Guide. Stamford, Connecticut; 1983 June; XSIG 038306.

Comprehensive tutorial on Interpress, intended for system designers and programmers writing creator software.

B

Appendix B Types, primitives, and standard vectors

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
Color	7	4.7	
Identifier	2	2.2.2	
Integer	–	2.2.1	subtype of Number
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 *MERGEPROP

Frames (§ 2.4.4):
FGET FSET

Operators (§ 2.4.5):
MAKESIMPLECO DO DOSAVE DOSAVEALL DOSAVESIMPLEBODY

Stack (§ 2.4.6):
POP COPY DUP ROLL EXCH MARK UNMARK UNMARK0 COUNT NOP

Control (§ 2.4.7)
IF IFELSE IFCOPY *COPYNUMBERANDNAME

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)
*MAKECOWITHFRAME *LASTFRAME *OBTAINEXTERNALINSTRUCTIONS *ADDINSTRUCTION-DEFAULTS *RUNSIZE *RUNGET

Imager state (§ 4.2):
IGET ISET *SETMEDIUM

Coordinate systems (§ 4.3):
*DROUND

Transformations (§ 4.4)
Making: *MAKET TRANSLATE ROTATE SCALE SCALE2 CONCAT
Current transformation and instancing: CONCATT MOVE TRANS SHOW SHOWANDXREL

Current position (§ 4.5):
SETXY SETXYREL SETXREL SETYREL GETCP

Pixel arrays (§ 4.6):
MAKEPIXELARRAY FINDDECOMPRESSOR

Color (§ 4.7):
MAKEGRAY FINDCOLOR MAKESAMPLEDBLACK SETGRAY

Masks (§ 4.8):
MOVETO LINETO LINETOX LINETOY MAKEOUTLINE MASKFILL MASKSTROKE MASKVECTOR MASKRECTANGLE STARTUNDERLINE MASKUNDERLINE MASKTRAPEZOIDX MASK-TRAPEZOIDY MASKPIXEL

Characters (§ 4.9):
FINDFONT MODIFYFONT SETFONT

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 this operator in the encoding.

VARIABLE STACK: it takes a variable number of arguments or returns a variable number of results.

BODY OPERATOR: it takes a body as its last argument.

REDUNDANT: it is an abbreviation for a *simple* Interpress program.

OPERATOR	SECTION	ENCODING VALUE	VARIABLE STACK	BODY OPERATOR	REDUNDANT
ABS	2.4.9	200			●
ADD	2.4.9	201			
*ADDINSTRUCTION-DEFAULTS	3.1	–			
AND	2.4.8	202			●
CEILING	2.4.9	203			●
*COMPUTECORRECTIONS	4.10	–			
CONCAT	4.4.3	165			
CONCATT	4.4.5	168			●
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	●		
DIV	2.4.9	204			
DO	2.4.5	231	●		
DOSAVE	2.4.5	232	●		
DOSAVEALL	2.4.5	233	●		
DOSAVESIMPLEBODY	2.4.5	120	●	●	●
*DROUND	4.3.4	–			
DUP	2.4.6	181			●
EQ	2.4.8	205			
*EQN	2.4.8	–			
EXCH	2.4.6	185			●
FGET	2.4.4	20			
FINDCOLOR	4.7	423			
FINDDECOMPRESSOR	4.6	149			
FINDFONT	4.9.1	147			
FLOOR	2.4.9	206			●
FSET	2.4.4	21			
GE	2.4.8	207			●
GET	2.4.3	17			
GETCP	4.5	159			
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	●	●	●

Xerox Private Data

OPERATOR	SECTION	ENCODING VALUE	VARIABLE STACK	BODY OPERATOR	REDUN-DANT
IGET	4.2	18			
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	–			
MAKEGRAY	4.7	425			
MAKEOUTLINE	4.8.1	417	●		
MAKEPIXELARRAY	4.6	450			
MAKESAMPLEDBLACK	4.7	426			
MAKESIMPLECO	2.4.5	114		●	
*MAKET	4.4.3	–			
MAKEVEC	2.4.3	283	●		
MAKEVECLU	2.4.3	282	●		
MARK	2.4.6	186	●		
MASKFILL	4.8.2	409			
MASKPIXEL	4.8.3	452			
MASKRECTANGLE	4.8.2	410			●
MASKSTROKE	4.8.2	24			
MASKTRAPEZOIDX	4.8.2	411			●
MASKTRAPEZOIDY	4.8.2	412			●
MASKUNDERLINE	4.8.2	414			●
MASKVECTOR	4.8.2	441			●
*MERGEPROP	2.4.3	–			
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			●
*OBTAINEXTERNAL-INSTRUCTIONS	3.1	–			
OR	2.4.8	213			●
POP	2.4.6	180			
REM	2.4.9	216			●
ROLL	2.4.6	184	●		
ROTATE	4.4.3	163			
ROUND	2.4.9	217			●
*RUNGET	3.3.3	–			
*RUNSIZE	3.3.3	–			
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	424			●
*SETMEDIUM	4.2	–			
SETXREL	4.5	12			●
SETXY	4.5	10			●

OPERATOR	SECTION	ENCODING VALUE	VARIABLE STACK	BODY OPERATOR	REDUN-DANT
SETXYREL	4.5	11			●
SETYREL	4.5	13			●
SHAPE	2.4.3	285			
SHOW	4.4.6	22	●		●
SHOWANDXREL	4.4.6	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			●

B.4 Non-redundant primitive operators

Name	*Section*	*Module (if not text)*
GET	2.4.3	
MAKEVECLU	2.4.3	
SHAPE	2.4.3	
*MERGEPROP	2.4.3	
FGET	2.4.4	
FSET	2.4.4	
DO	2.4.5	
DOSAVE	2.4.5	
DOSAVEALL	2.4.5	
MAKESIMPLECO	2.4.5	
POP	2.4.6	
COPY	2.4.6	
ROLL	2.4.6	
MARK	2.4.6	
UNMARK	2.4.6	
COUNT	2.4.6	
IF	2.4.7	*Computation*
IFCOPY	2.4.7	*Computation*
*COPYNUMBERANDNAME	2.4.7	
EQ	2.4.8	*Computation*
GT	2.4.8	*Computation*
TYPE	2.4.8	*Computation*
ADD	2.4.9	*Computation*
SUB	2.4.9	*Computation*
MUL	2.4.9	*Computation*
DIV	2.4.9	*Computation*
TRUNC	2.4.9	*Computation*
*LASTFRAME	3.1	
*MAKECOWITHFRAME	3.1	
*OBTAINEXTERNALINSTRUCTIONS	3.1	
*ADDINSTRUCTIONDEFAULTS	3.1	
IGET	4.2	
ISET	4.2	
*SETMEDIUM	4.2	
*DROUND	4.3.4	
CONCAT	4.4.3	
*MAKET	4.4.3	
ROTATE	4.4.3	
GETCP	4.5	
MAKEPIXELARRAY	4.6	*Binary*
FINDDECOMPRESSOR	4.6	*Binary*
MAKEGRAY	4.7	*Gray*
FINDCOLOR	4.7	*Gray*
MAKESAMPLEDBLACK	4.7	*Gray*

Name	*Section*	*Module (if not text)*
MOVETO	4.8.1	*Polygons*
LINETO	4.8.1	*Polygons*
MAKEOUTLINE	4.8.1	*Polygons*
MASKSTROKE	4.8.2	*Polygons*
MASKFILL	4.8.2	*Polygons*
MASKPIXEL	4.8.3	*Binary*
FINDFONT	4.9.1	
MODIFYFONT	4.9.2	
CORRECT	4.10	
CORRECTMASK	4.10	
CORRECTSPACE	4.10	

B.5 Standard vectors

The following vectors and property vectors in Interpress have defined meanings of their elements.

Name	*Section*	*Structure*
instructions	3.3.3	[*breakPageFont*, Vector, *docName*, BreakPageString, *docCreator*, BreakPageString, *docComment*, BreakPageString, *docCreationDate*, BreakPageString, *docPassword*, Vector, *jobSender*, BreakPageString, *jobRecipient*, BreakPageString, *jobStartMessage*, BreakPageString, *jobEndMessage*, BreakPageString, *jobStartWait*, Integer, *jobEndWait*, Integer, *jobAccount*, Any, *jobPriority*, Identifier, *jobErrorAbort*, Identifier, *jobSummary*, Identifier, *jobPassword*, Vector, *finishing*, Identifier or Vector, *plex*, Identifier, *media*, Vector of MediumDescription, *copySelect*, Run of Integer, *copy Name*, Run of Integer, *onSimplex*, Run of Integer, *pageSelect*, Run of Run of Integer, *mediaSelect*, Run of Run of Integer]
MediumDescription	3.3.3	[0: Identifier or Vector of Identifiers (*name*), 1: Number (*mediumXSize*), 2: Number (*mediumYSize*)]
FontDescription	4.9	[*operators*, Vector, *characterMetrics*, Vector, *metrics*, Vector (Metrics), *name*, Vector]
CharacterMetrics	4.9	[*widthX*, Number, *widthY*, Number, *amplified*, Integer, *correction*, Integer, *leftExtent*, Number, *rightExtent*, Number, *descent*, Number, *ascent*, Number, *centerX*, Number, *centerY*, Number, *kerns*, Vector of [0: Integer, 1: Number, 2: Number], *ligatures*, Vector of [0: Integer, 1: Integer], *superscriptX*, Number, *superscriptY*, Number, *subscriptX*, Number, *subscriptY*, Number]
Metrics	4.9	[*easy*, Vector, *xHeight*, Number, *slant*, Number, *underlineOffset*, Number, *underlineThickness*, Number]

Although imager variables are not represented as a vector, they are accessed with Integer indices. These indices are summarized in Table 4.1, § 4.2.

Appendix C
Interpress name registry

Organizations wishing to obtain universal names that can be referenced reliably from any Interpress master should apply for a name in the Interpress registry by contacting:

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

D

Appendix 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.

Glossary

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 *width* 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 (5.3)

approximation: finding an *external* font which is close to the one requested, but not necessarily identical (4.9.1)

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 (3.3.2)

char: abbr. for character

character index: an Integer, sometimes called a "character code," that identifies a particular character; used to index a *font* (4.9)

character coordinate system: a standard coordinate system in which each *character operator* is defined (4.9)

character operator: a *composed operator* which, when executed, defines a character's *mask* (4.9)

co: abbr. for *composed operator*

color: the specification of the color (absorption coefficient of the ink) with which to show a primitive image (4.1, 4.7)

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 *widths* 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: abbr. 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*, *decompression operators* (3)

Xerox Private Data

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: abbr. for *frame*

font: a *vector* of *operators* designed to produce images of *symbols* (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 (3.2.2)

hierarchical name: a *vector* of *identifiers* that represent a structured name (3.2.2)

ICS: abbr. for *Interpress coordinate system*

id: abbr. 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: abbr. for *instructions.*

instance: usually refers to an image on the page of a standard *symbol.* For example, the word SHIPS when printed, contains two instances of the symbol S (4.4)

instructions: abbr. for *printing instructions*

instructions body: an optional *body* in a *master* that contains *printing instructions* (3.)

int: abbr. for *integer*

integer: a mathematical integer in a limited range; one of the types of the *base language* (2.2.1)

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 (5.1)

limited imager: an *imager* that cannot handle pages of arbitrary complexity (5.1.3)

literal: a representation in a master of a 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)

metric master: an encoding of all of a printer's font *metrics,* used by a *creator* to prepare a *master* (4.9)

metrics: of a character or *font,* the measurements of its critical dimensions (4.9)

mica: a unit of distance equal to 10^{-5} meter (4.3)

name: an *integer* 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: abbr. 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* (3.1.2)

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 (3.3.4)

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

point size: when referring to the size of a character, the normal spacing between lines of type of that size, e.g., 10-point type is sized so that it will be most legible when printed in lines spaced 10 points apart (4.9)

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*

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)

subset: a rough characterization of the capabilities of an Interpress *printer* (5.1)

symbol: a graphical shape; several *instances* of a symbol may appear in a *page image* (4.4.1)

text: the lowest *subset* of Interpress, which every *printer* is required to implement (5.1)

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 name: a *name* defined in the Interpress universal registry (3.2.2)

universal property vector: a *property vector* that can be extended using property names that are *hierarchical names* (2.4.3)

unlimited imager: an *imager* that can print pages of arbitrary complexity (5.1.3)

upper bound: the integer name of the last *element* of a *vector* (2.2.4)

variable: part of the imager state (4.2)

vec: abbr. for *vector*

vector: a sequence of values named by Integers (2.2.4)

width: of a character, the spacing from one character to the next (4.9)

Index

Xerox Private Data

Xerox Private Data

XEROX

Xerox Corporation
Stamford, Connecticut 06904

Printed in U.S.A.