AMT Datasouth Fastmark

PAL™ Print and Program Language Reference Guide

P/N 108744 Rev. A1

PUBLISHED BY
AMT Datasouth

4216 Stuart Andrew Boulevard

Charlotte, North Carolina 28217

Phone: 704.523-8500

Service 800.476.2450

Sales: 800.476.2120

Internet: www.amtdatasouth.com

Revised 4 December, 2003.

Contents

1 . Introduction

Welcome to the world of the PAL™ Print and Program Language. Since PAL™ is both a powerful printing language and programming language, many print applications not previously possible are now within reach.

Now you don't have to get a different software-specific printer for every application in your facility. AMT Datasouth PAL™ enabled printers such as the Fastmark™ line can translate, filter, interpret and understand almost any data stream. You can now replace obsolete devices with cost-effective, high-print-quality thermal printers -- with no expensive software changes.

Here is a brief overview:

- Quick, high-resolution printing of labels, tags and more
- Features PAL™ (no host/PC software reprogramming required!)
- Works for all departments, regardless of what software they're using
- Can be run even when the system is down (label format is stored in printer)
- Offers plug and play convenience
- Barcodes can be added to existing print jobs without changing host/PC software

How does PAL™ enable you to print your current data stream—without reprogramming your system?

As documented in this manual, PAL™ is not only a powerful printing language, it is also a fully functional programming language. Utilizing this manual, powerful PAL™ programs may be written and loaded in the printer to perform a variety of tasks such as interpreting legacy data steams. Or we can prepare a PAL program for each of your applications and pre-load it into your PAL™ enabled printer such as Fastmark™ at the factory. These programs enable the PAL™ enabled printer to print labels by filling in the variable fields using your current data stream. This data stream could have been intended for a laser, dot matrix, ink jet or embossing printer. And, you don't have to worry about the software driver in the host, because the formatting is all done within the printer.

PAL™ enabled printers such as Fastmark are packed with features and options such as:

- Parallel and serial ports
- Optional Ethernet, Twinax, Coaxial and USB communication
- Extensive on-board complement of linear and 2-D symbology barcodes
- Smooth scalable fonts
- Expandable memory options
- Flash memory drives for storing PAL™ programs and other data.
- Optional external keyboard for stand-alone applications
- SRAM and Flash memory cards
- Optional real-time clock (RTC) with Flash memory
- Optional Peel and Present with label taken sensor
- Optional Cutters
- Rugged cabinet construction

All PAL™ enabled products include the latest in Windows™ drivers.

To learn more about how the PAL™ Print and Program Language can be used to quickly and cost effectively integrate PAL™ enabled printers such as Fastmark™ into your facility, call AMT Datasouth at 800-215-9192.

2. PAL Fundamentals

2.1. The PAL Interpreter

Every PAL printer contains a copy of the PAL interpreter. The PAL interpreter is the software inside the PAL printer which the printer's internal computer executes. Although the PAL interpreter serves a different purpose, it is essentially an application program just like any word processing or spreadsheet application program run on a general purpose computer.

2.2. Sending Data to PAL Printers

Word processing and speadsheet applications usually accept their data from a user sitting at a keyboard. The PAL interpreter usually accepts its data from a host computer via the printer's electrical interface with the host computer. Word processors and spreadsheets can also accept data from other sources like files located on the user's disk drive. The PAL interpreter can also accept data from other sources like memory cards plugged into the printer.

The data supplied to the PAL interpreter consists of a series of human readable characters. The PAL interpreter does not require any special control characters which only computers can understand.

2.3. PAL Objects

When a host computer sends data to a PAL printer, the PAL interpreter software inside the printer receives the data and analyzes it. First, the interpreter groups the series of characters into individual objects. Each object represents a single piece of data for the interpreter. Therefore, the interpreter views the data it receives as a series of data objects.

The interpreter separates the series of characters into objects by looking for one or more spaces or other special character. PAL recognizes the following characters as being the same as a space. PAL refers to all of these characters as *whitespace* characters.

Character	ASCII Code	Octal Value	Hexadecimal Value
Space	SP	040	20
Tab	HT	011	09
Carriage Return	CR	015	0D
New Line / Line Feed	LF	012	0A
Null	NUL	000	00
Form Feed	FF	014	0C

PAL requires the user to separate each object with at least one of the preceeding whitespace characters, or one of the following special characters.

Character	Description	Octal Code	Hexadecimal Code
(	Left/Open Parenthesis	050	28
)	Right/Close Parenthesis	051	29
<	Left/Open Angle Bracket or Less Than	074	3C
>	Right/Close Angle Bracket or Greater Than	076	3E
[	Left/Open Square Bracket	133	5B
]	Right/Close Square Bracket	135	5D
{	Left/Open Brace	173	7B
}	Right/Close Brace	175	7D
/	Forward Slash	057	2F
%	Percent	045	25

The special characters listed above each have a special meaning for PAL. The user should only use one or more of these characters to separate objects when the user also wishes PAL to perform the action associated with the character.

2.4. Interpreter Operation

Normally the PAL interpreter performs a function in response to each object received. For data objects, PAL usually just stores the object within the printer's memory. Executable objects instruct PAL to perform some operation. The operation usually involves one or more of the data objects previously received.

The interpreter immediately performs the appropriate function for each object upon receipt of the object. However, PAL does not consider an object fully received until it receives the separation character which follows the object. Therefore, if the host computer sends the printer a command without following the command with a space or other separation character, the printer will not respond to the command until it receives the separation character. Until PAL receives the separation character, PAL cannot know for sure whether or not it has received all the characters of the object.

2.5. Operand Stack

As PAL receives data objects from the host, it *pushes* the objects onto an internal structure known as the *operand stack*. PAL places each successive object on top of the previous object on this stack of objects. As long as the interpreter continues to receive data objects, it will continue pushing the objects onto this stack.

When PAL receives an object which indicates some action for the interpreter to perform, the action will usually involve zero or more data objects know as *operands* or *parameters*. For example, a div (divide) operation requires two operands — the divisor and the dividend. In order to perform the operation, PAL *pops* the top two operands off the operand stack. It then divides one operand by the other operand in order to calculate the quotient. PAL then pushes the quotient onto the operand stack.

2.6. Post-Fix Notation

PAL receives data objects and operation objects in an order known as post-fix notation. This means that the data upon which an operator will operate occurs before the operator itself. This differs from the algebraic notation which everyone learned in school.

In school, algebraic equations looked like the following.

(1 + 2) × (4 + 5)

In post-fix notation, this same equation has the following format.

1 2 + 4 5 + ×

Post-fix notation has the advantage that it does not require parethesis or other special symbols to override operator precedence. The preceding algebraic equation required parenthesis it order to give the addition operations precedence over the multipication operation. In post-fix notation, the operators have no implied precedence. The left-most operation occurs first, and the right-most operator occurs last. Therefore, both the computer and human need only perform the equation from left to right as written.

In order to perform very complex equations, both algebraic and post-fix notation rely upon an operand stack. When humans perform complex algebraic calculations manually, they imitate an operand stack using a piece of scratch paper. When using an algebraic calculator, the calculator contains an internal operand stack. The "(" and ")" keys on the calculator instruct the calculator to perform push and pop operations. As the equations above show, an algebraic calculation would require 12 key presses, not including the final "=" key, on a calculator.

Hewlett-Packard calculators have been based on *reverse polish notation* (or "RPN" for short) for years. Reverse polish notation operates the same as post-fix notation. The post-fix notation equation shown above also shows seven of the nine key strokes necessary to perform the calculation using a Hewlett-Packard RPN calculator. The equation does not show the necessary "enter" key press following the 1 and 4.

Although not as familiar to us as algebraic notation, post-fix notation actually provides a faster and more straight-forward approach to performing calculations.

PAL performs all of its operations using post-fix notation. As PAL encounters data objects, it automatically pushes the objects onto the operand stack. When PAL encounters an object which indicates an action to perform, PAL pops any necessary operands from the operand stack, performs the action, and pushes any results onto the operand stack. In PAL, the preceding equation would appear as the following sequence of PAL objects.

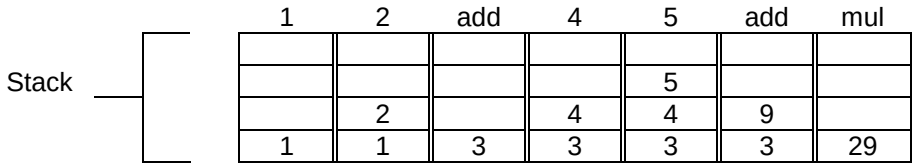
```
1 2 add 4 5 add mul
```

When PAL encounters the data object "1," it pushes the object onto the operand stack. PAL performs the same action for the data object "2." PAL then encounters the operator "add." In response, the interpreter pops the "2" and the "1" off the stack and adds them together. This produces the result data object "3," which PAL pushes onto the stack.

Next PAL encounters the data objects "4" and "5," which it pushes onto the stack. Then PAL encounters the operator "add," so PAL pops the "4" and "5," adds them together, and pushes the resulting "9". Finally, PAL encounters the "mul" operator. In response, PAL pops the "9" from the

second "add" and the "3" from the first add and multiplies them together. It then pushes the result, "27," onto the operand stack.

The following diagram shows the contents of the operand stack when PAL finishes processing each object in the above sequence.



2.7. systemdict, globaldict, userdict

PAL supports a data structure called a *dictionary*. Dictionaries contain an arbitrary collection of objects organized into pairs. PAL treats the first object of each pair as the *key* object, and the second object as the *value* object. This organization allows PAL to search a dictionary for a particular key object, and recover the value object associated with the key.

When the PAL interpreter initializes, it automatically creates three dictionaries to help it keep track of the various objects a PAL programmer will store into the printer's memory. PAL gives these dictionaries the names systemdict, globaldict, and userdict.

Initially, userdict and globaldict do not contain any objects. systemdict contains all the names which PAL recognizes as operators, as well as other predefined names like true, false, and mark.

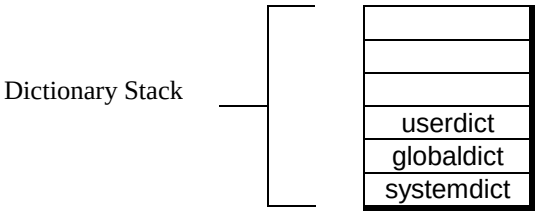
The operator names themselves do not actually instruct the PAL interpreter to perform an operation. Instead, PAL has a special internal object type known as an intrinsic operator object. When the PAL interpreter encounters one of these intrinsic operator objects, the interpreter performs the action indicated by the object.

systemdict contains all the operator names as key objects. It also contains all the intrinsic operator objects as value objects associated with the appropriate name objects. Therefore, when PAL receives an executable name object from the host computer, PAL searches systemdict to find a dictionary key entry which matches the name object.

When PAL locates the matching name, the interpreter then recovers the value object associated with the name. For the operator names, PAL will find an intrinsic operator object associated with the name. The PAL interpreter then performs the action indicated by the intrinsic operator object.

2.8. Dictionary Stack

During initialization, PAL pushes the three standard dictionaries, systemdict, globaldict, and userdict onto an internal structure known as the *dictionary stack*. Therefore, immediately after initialization, the dictionary stack contains the following.



When PAL encounters an executable name, PAL goes to the dictionary stack to find out what to do. PAL starts by trying to locate the name in the top-most dictionary on the dictionary stack. If it cannot find the name, it then tries the next dictionary down on the stack. PAL continues down the stack until it locates the name. Once PAL locates the name, it stops searching.

Since PAL stops searching when it locates the name, any entries for a given name in a dictionary on the top of the dictionary stack will supercede an entry in a dictionary on the bottom of the stack. As the above diagram shows, systemdict resides on the very bottom on the dictionary stack. Therefore, any entry for a name in any other dictionary on the dictionary stack will have precedence over the entry for that name in systemdict.

As a result, the programmer has the freedom to redefine any of the names which PAL has predefined for performing the various PAL operations. However, redefining PAL operators only serves to make a PAL order sequence difficult for another programmer to understand.

The dictionary stack also serves a more important purpose. It allows the PAL programmer to define new names to which the PAL interpreter will automatically respond. The programmer can add a new name with an associated value object to one of the dictionaries on the dictionary stack. Later, when PAL encounters the name, it will search the dictionary stack and find the programmer's entry.

If the programmer associates a procedure object with the name, PAL will automatically execute the procedure. If the programmer associations an integer, string, or other data object with the name, PAL will automatically push the data object onto the operand stack.

The programmer may not alter the contents of systemdict. However, PAL automatically provides the programmer with the two dictionaries userdict, and globaldict. The programmer may freely add and delete entries from these dictionaries. During initialization, PAL creates userdict and globaldict as empty dictionaries.

PAL also provide operators which allow the programmer to push new dictionaries onto the dictionary stack and later pop them off the stack. This allows the programmer to collect localized definitions into separate dictionaries and then discard all the definitions by simply popping the dictionaries from the dictionary stack.

Although PAL allows the programmer to add and delete entries within userdict and globaldict, PAL does not allow the programmer to remove the three standard dictionaries themselves from the dictionary stack.

2.9. Virtual Memory

The interpreter keeps all user data objects as well as the various interpreter data structures within the *virtual memory* area. PAL refers to this memory area as *virtual* memory because the programmer does not have direct access to this memory.

PAL dynamically manages this space for the programmer. As the programmer sends objects to the PAL interpreter, the interpreter automatically allocates space for the objects within the virtual memory area. When the programmer no longer requires a particular object, PAL automatically frees the object's memory for use by other objects.

PAL will keep an object within virtual memory for as long as the programmer maintains a reference to the object. Once the programmer eliminates all references to the object, PAL automatically removes the object from memory.

The programmer still has references to a given object so long as the programmer still has some means of acessing the object. For example, object A may contain the only reference anywhere in memory to the object B. In turn, the object B may contain the only reference anywhere in memory to the object C. And the operand stack may contain the only reference to object A.

So long as a reference remains anywhere in memory to the object A, PAL will keep all of these objects in the virtual memory. However, if the programmer pops the reference to object A from the operand stack without creating an alternate reference to the object, the programmer will have eliminated all references to object A. As a result, PAL will eliminate object A from memory.

When PAL eliminates object A from memory, it also eliminates the only reference to object B. Therefore, PAL also eliminates object B from memory. This results in the elimination of the only reference to object C, so PAL eliminates it from memory as well.

The amount of virtual memory available to the programmer varies between PAL printer models. In can also vary between two different printers with the same amount of internal memory. This variance results from the amount of memory the PAL interpreter requires in order to manage the various options on different printer models.

PAL provides the vmstatus operator to allow the programmer to determine the amount of virtual memory available on a given printer model. The operator also provides information relating to the amount of virtual memory already allocated for PAL objects and other data.

2.10. Transformation Matrix

All printers provide some form of coordinate system. The coordinate system provides the basis for the user to instruct the printer where to locate a particular character or other image on the page. Many non-PAL printers base their coordinate system on the printer's dots-per-inch or dots-per-millimeter resolution.

This works fine so long as the host computer programmer must only control that particular printer model. However, if the programmer must also control other printers which use different resolutions, then the same control sequences will not work in all cases, even if all printers use the same basic control language.

PAL's device independent coordinate system allows the host programmer to use the same control sequence for all PAL printers. This includes PAL printers with different resolutions as well as from different printer manufacturers.

In addition to providing a device independent coordinate system, PAL allows the programmer to define this coordinate system to meet the needs of the programmer. By default, PAL uses a typesetters' unit of measure known as a *point*. No precise definition of a point exists, however typesetters generally use values close to 1/72 of an inch. Most computer software, including PAL, use exactly 1/72 of an inch as the definition of a point.

PAL realizes that a point may not suit every programmer's requirements. Therefore, PAL provides operators which allow the programmer to alter the current coordinate system. The programmer can freely scale, rotate, and relocate the origin of the user coordinate system.

In order to convert the user's coordinates to dots on a printed page, PAL maintains an internal mathematical construct known as a *transformation matrix*. The transformation matrix contains six values which PAL changes whenever the user alters the coordinate system. In mathematics, a

transformation matrix also includes three additional constant values. However, since the extra values do not change, PAL does not need to keep the values as part of the matrix.

The following diagram shows the mathematical representation of a transformation matrix.

$$\begin{bmatrix} A & B & 0 \\ C & D & 0 \\ E & F & 1 \end{bmatrix}$$

Special mathematical rules exist for changing the values A through F in response to scaling, rotating, and relocating the origin (translating) of a coordinate system. However, once PAL has updated the six values to reflect any changes to the user's coordinate system, PAL simply uses these values in the following formulas to convert the user's coordinates to actual dot positions on a printed page.

$$X' = AX + CY + E$$
$$Y' = BX + DY + F$$

X and Y represent a coordinate in the user's coordinate system. X' and Y' represent the same coordinate on a printer page.

PAL uses the term *current transformation matrix* to indicate the transformation matrix currently in use by PAL. PAL automatically initializes the current transformation matrix to the values necessary to convert the PAL default coordinate system (points) to the physical page coordinate system (dots).

The discussions within this manual of the various PAL operators which affect the current transformation matrix describe the various ways in which the programmer may alter the matrix.

3. Objects

PAL allows programmers to store various different types of data into the printer's memory. PAL uses the term *object* to refer to each different piece of data stored within the printer's memory. Each object has a type. An object's type indicates how PAL will interact with that particular object.

The PAL language groups the various object types into two classifications — simple and composite. In addition, PAL includes a classification of object types internally used by PAL. This manual discusses each object type under its appropriate classification.

3.1. Simple Objects

Simple objects represent the basic types of data which the programmer can store within the printer. This differs from composite objects which group together collections of simple objects as well as other composite objects. The simple object classification includes the following types:

Integer
Fixed-Point
Boolean
String
Name
Mark
Null

3.1.1. Integer Objects

PAL allows integer objects to have numerical values between -999,999,999 and +999,999,999, inclusive. Integer objects cannot have any fractional digits. In order words, an integer object cannot have the value 1.5.

When the programmer includes an integer value as part of a PAL sequence, the value can include only the digits 0 through 9 with an optional leading plus (+) or minus (-) sign. If the programmer does not specify a plus or minus sign, PAL assume a positive value. Integer values may not include a decimal point even if the programmer places only zeros to the right of the decimal point. The value also may not include commas or other punctuation.

The following PAL sequence specifies three integer objects for the PAL interpreter.

```
-45 +36 999
```

3.1.2. Fixed-Point Objects

PAL allows fixed-point objects to have numerical values between -999,999,999.999,999,999 and +999,999,999.999,999,999, inclusive. This means that fixed-point objects may have nine digits to the left of the decimal point, and another nine digits to the right of the decimal point.

PAL differs from many other programming languages in its use of fixed-point values. Most other programmer languages use floating-point values. Floating-point values usually allow the programmer to specify a small number of significant digits, but the digits may have almost any relationship to the decimal point. For example, many programming languages allow around six

significant digits for their floating-point values. However, these six digits can represent the value 1 trillion (1,000,000,000,000) or 1 trillionth (0.000,000,000,001).

Floating-point values work very well in scientific applications. For example, specifying the distance to a star or the size of an atom. However, they do not work very well in business applications. Business applications tend to require a smaller range of values, but many more significant digits. Few companies have the need to calculate their worth in the billions. And those companies which do can afford super computers to count their money for them.

However, many companies require calculations in the tens of millions or less, with every digit being significant. Floating-point values generally cannot keep track of sufficient digits to satisfy this requirement. Therefore, PAL relies upon large fixed-point numbers instead of more conventional floating-point values.

When the programmer includes a fixed-point value as part of a PAL sequence, the value must have a particular format. The value may start with an optional plus (+) or minus (-) sign. If the programmer does not include a plus or minus sign, PAL will assume a positive value. The value must also have a digit, 0 through 9, both *before and after* a decimal point. When PAL sees the decimal point, PAL knows to treat the value as a fixed-point value rather than an integer value.

The value may not include commas or other punctuation. If the programmer does not include a digit both before and after the decimal point, PAL will treat the value as a name object rather than a fixed-point object. For example, PAL treats "1." and ".1" as name objects. The programmer must specify "1.0" or "0.1" in order for PAL to treat the objects as fixed-point numbers.

3.1.3. Boolean Objects

Boolean objects can only have the value true or false. PAL usually creates boolean objects in response to performing some test. For example, if the programmer instructs PAL to test two integers for equality, PAL will create a boolean object which indicates the result of the test. If PAL finds the integers equal, PAL will create a boolean object with the value true. If PAL does not find the integers equal, PAL will create a boolean object with the value false.

PAL also includes definitions for the names true and false. The name true corresponds with a boolean object having the value true. The name false corresponds with a boolean object having the value false.

3.1.4. String Objects

Strings consists of a variable length collection of bytes. In simple applications, each byte usually represents a printable character. A string can contain from zero to 30,000 bytes.

Since each string object can have a variable number of bytes associated with it, PAL stores the string object and the collection of bytes (the string value) in separate parts of memory. The string object contains only a reference to the string value.

When the programmer instructs PAL to perform an operation which causes PAL to duplicate a string object, PAL only duplicates the object part of the string. PAL does not duplicate the value portion of the string. As a result, the duplication creates two objects which both refer to the same collection of bytes. Special operators exist which allow the programmer to instruct PAL to also duplicate the value portion of the string.

In many cases, the programmer will not wish PAL to duplicate the value portion. Duplicating only the object portion of the string does not consume very much of the printer's memory. On the other hand, duplicating the value portion of a large string will consume a large amount of the printer's memory.

PAL accepts strings as text enclosed in parenthesis. For example, "(hello)" specifies a string consisting of the characters "h," "e," "l," "l," and "o."

PAL also allows the programmer to include parenthesis as part of the string. Strings containing balanced pairs of parenthesis do not require any special treatment. For example, PAL also accepts "(ab(cd)ef)" as a perfectly valid string. In this case, the string contains eight characters — "ab(cd)ef."

If the string contains unbalanced parenthesis, then the programmer should place the special back-slash (\) character in front of each parenthesis. The programmer may also use a back-slash even when the string contains balanced parenthesis. A computer program which generates the strings to send to a PAL printer would normally just place a back-slash in front of every parenthesis. Therefore, the programmer could also specify the preceeding string as "(ab\(cd\)ef."

In order to include the back-slash as part of a string, the programmer need only specify two back-slashes. For example, specifying "(The back-slash character \(\\\) is a prefix!)" creates the string "The back-slash character (\) is a prefix!"

The following table list all the special characters which the programmer can place into strings using the back-slash character.

PAL Code	Description	ASCII Symbol	Octal Value	Hexadecimal Value
\n	New Line	LF	012	0A
\r	Carriage Return	CR	015	0D
\t	Tab	HT	011	09
\b	Backspace	BS	010	08
\f	Form-Feed	FF	014	0C
\\	Back-Slash	\	134	5C
\(	Left (Open) Parenthesis	(	050	28
\)	Right (Close) Parenthesis	)	051	29
ddd	Character for Octal Code *ddd*	*any*	*ddd*	

3.1.5. Name Objects

Just like strings, names consist of a variable length collection of bytes. In simple applications, each byte usually represents a printable character. A name can contain from one to 30,000 bytes.

Since each name can have a variable number of bytes associated with it, PAL stores a name object in exactly the same manner as it stores string objects. In fact, PAL treats name objects and string objects in an almost identical manner.

Each name objects has one of three different attributes — *executable*, *literal*, or *immediate evaluation*. The programmer specifies which attribute the name object should have by placing zero, one, or two forward slashes (/) immediately in front of the name.

If the programmer does not place any forward slash in front of the name, PAL treats the name as executable. For example, PAL will treat the character sequence "MyName" as an executable name. Provided PAL does not encounter the name while creating a procedure, PAL will try to find

an object or an operation associated with the name. If PAL encounters the name while creating a procedure, PAL does not execute the name at that time. PAL simply stores the name as part of the procedure. PAL will execute the name later when the programmer instructs PAL to execute the procedure.

If the programmer places a single forward slash in front of the name, PAL treats the name as literal. For example, PAL will treat the character sequence "/MyName" as a literal name. This means that PAL will simply create a name object in a manner similar to a string object.

If the programmer places two forward slashes in front of the name, PAL immediately evaluates the name. For example, PAL will treat the character sequence "//MyName" as an immediately evaluated name. When PAL *evaluates*, as opposed to *executes*, a name, PAL simply replaces the name with the object associated with the name. PAL does not attempt to execute the object associated with the name.

Except during a procedure definition, the difference between executable and immediately evaluated only matters for names associated with procedures or PAL intrinsic operators. In all other case, executable and immediately evaluated names produce the same results.

During a procedure definition, PAL simply records executable names as part of the procedure. It does not attempt to execute the name at that time. However, PAL still substitutes immediately evaluated names with their associated objects even during procedure definitions.

Using an immediately evaluated name during a procedure definition can produce entirely different results from using an executable name. For example, the programmer has associated the name FirstProc with a procedure object. The programmer then defines a second procedure, SecondProc, which includes a reference to FirstProc.

If the programmer specifies FirstProc as an executable name (no "/" in front of the name), PAL will simply record the executable name FirstProc within SecondProc's definition. Later, when the programmer instruct PAL to execute SecondProc, PAL will also execute the procedure associated with the name FirstProc when it encounters the executable name.

If the programmer changes the procedure associated with FirstProc between executions of SecondProc, PAL will always execute the procedure associated with FirstProc at the time PAL executes SecondProc. It will not matter which procedure was associated with FirstProc at the time SecondProc was defined. In fact, PAL will not care whether the programmer has associated any procedure with FirstProc before defining SecondProc. PAL only cares about the procedure associated with FirstProc's at the time PAL encounters FirstProc when executing SecondProc.

If the programmer specifies FirstProc as an immediately evaluated name (//FirstProc) within the definition of SecondProc, PAL will immediately substitute //FirstProc with the current object associated with FirstProc. This means that PAL will place the procedure object associated with FirstProc directly within the definition of SecondProc.

PAL will insert the procedure *object* and not the instructions contained within the procedure object. This will have the same effect as having used the "{" and "}" operators to define the procedure directly within SecondProc. When PAL encounters the procedure object while executing SecondProc, PAL will simply push the procedure object onto the operand stack. PAL will not automatically try to execute the instructions contained within the procedure object.

Names can include almost any combination of characters, numbers, and special symbols. However, names do not have special enclosing characters like the parenthesis required by strings. This means

that names cannot include the special characters which PAL uses for other purposes. Specifically, names cannot include any of the following object separator characters.

( ) < > [ ] { } / %

Also, a name cannot satisfy the rules for an integer or fixed-point object. Otherwise, PAL will treat the name as an integer or fixed-point object. Therefore, PAL accepts "1+" as a name, but "+1" as an integer object. Likewise, PAL accepts ".1" and "1." as names, but "0.1" and "1.0" as fixed-point objects.

3.1.6. Mark Objects

A mark object does not have a value. It only has the type *mark*. Mark objects serve a special purpose under PAL. Several PAL operators manipulate all objects pushed onto the operand stack above a mark object. PAL has predefined the name mark and has associated the name with a mark object.

3.1.7. Null Objects

PAL uses null objects as place holders. For example, when the programmer instructs PAL to create an array object, PAL fills the array object with null objects. The null objects act as place holders until the programmer replaces them with other objects.

3.2. Composite Objects

In addition to simple objects, PAL supports three types of composite objects — arrays, dictionaries, and procedures.

Composite objects group together collections of other objects. A composite object may contain any combination of simple objects as well as other composite objects. One composite object may contain numerous other composite objects, which in turn contain numerous other composite objects, which in turn contain numerous other composite objects. PAL does not impose any limitation on the complexity of combinations which the programmer can create.

3.2.1. Array Objects

Array objects simply contain a list of other objects. Unlike most other programming languages, array objects may contain any combination of object types. For example, an array might contain a couple integer objects, four other array objects, six dictionary objects, three string objects, four name objects, and six procedure objects.

PAL provides three operations for creating array objects. The first operator, array, creates an array containing all null objects. This allows the programmer to create an array and place the data into it at a future time. The PAL sequence "24 array" creates an array containing 24 null objects.

The other two PAL operators, "[" and "]," work as a team to create an array containing a desired collection of objects. The first operator of the pair, "[," starts the array definition. The "[" operator does nothing more than place a mark object onto the operand stack. In fact, PAL does not care whether the programmer uses the "[" operator or the predefined name mark to place the mark object onto the stack. However, using the "[" operator makes PAL sequences much easier for humans to read.

Unlike most other programming languages, PAL does not treat the "[" and "]" operators as special language syntax symbols. PAL does not treat the operators and data it encounters between the "[" and "]" operator any different than if it had not encountered the "[" operator. In fact, once PAL places the mark object onto the stack in response to the "[" operator, PAL totally forgets that it ever saw the "[" operator.

PAL creates the array object in response to encountering the "]" operator. PAL creates the array from all the objects located on the operand stack above the top-most mark object. Normally, the top-most mark object will result from the previous "[" operator.

Therefore, the programmer need only push all the objects for the array onto the operand stack following the mark object pushed by the "[" operator. PAL does not care how the programmer pushes the objects onto the stack. In the most simple case, the programmer may simply list all the objects. PAL will push the objects onto the stack as part of PAL's normal duties. In a more complex case, the programmer may execute any combination of procedures and PAL operators to generate the data for the array.

For example, the following simple PAL sequence creates an array containing the integer object 123, the string object "hello," and the literal name object "MyName."

```
[123 (hello) /MyName]
```

PAL treats this sequence in a very straight-forward manner. When PAL encounters the "[" operator, it pushes a mark object onto the operand stack. PAL then encounters the integer object "123" and pushes it onto the stack. Next PAL encounters the string object "(hello)" and pushes it onto the stack. After the string, PAL encounters the literal name object "/MyName" and pushes it onto the stack.

Finally, PAL encounters the "]" operator. This instructs PAL to create an array object from all of the objects on the stack above the top-most mark object. As a result, PAL creates an array containing the integer, string, and name objects. PAL then removes the objects, as well as the mark object, from the operand stack and places the array object onto the stack.

The following diagram shows the organization of the above array within the printer's memory.

Index	Object
0	123
1	(hello)
2	/MyName

Like string and name objects, array objects actually consist of two parts — the object part and the value part. The array object itself contains only a reference to the array data (value). Therefore, when the programmer instructs PAL to duplicate an array object, PAL only creates a new reference to the array data. PAL does not duplicate the data itself. This conserves memory and allows the programmer to manipulate the array data in various ways.

The programmer may access the individual objects within the array by specifying the index of the object. The first object in the array has an index of zero, the next object has an index of one, with the indexes continuing through N-1, where N represents the number of objects in the array. Therefore, an array has the following general appearance.

[*obj0 obj1 obj2 obj3 ... objN-1***]**

3.2.2. Dictionary Objects

The programmer creates dictionary objects in the exact same manner as arrays. Except, dictionary objects use the "<<" and ">>" operators. The "<<" operator serves the exact same purpose as the "[" operator and mark predefined name. When PAL encounters the "<<" operator, PAL simply pushes a mark object onto the stack.

Later, when PAL encounters the ">>" operator, PAL builds a dictionary object from all the objects on the stack above the top-most mark object. Between the "<<" and ">>" operators, the programmer may perform any combination of operations necessary to place the desired dictionary data onto the stack.

PAL organizes the entries within a dictionary into pairs. PAL treats the first object of each pair as a *key*, and the second object as the *value* associated with the key. Therefore, the programmer must always specify an even number of objects when creating an array.

When pushing dictionary objects onto the operand stack when creating a dictionary, the programmer must first push the key object of each pair followed by the value object. Therefore, a dictionary has the following general appearance.

<<**key0 value0 key1 value1 key2 value2 key3 value3 ... keyN-1 valueN-1**>>

Dictionaries allow the programmer to access a particular *value* object by specifying the *key* object associated with the value. This provides a very powerful mechanism for organizing data.

Like array objects, PAL allows dictionary objects to contain any arbitrary combination of object types. Although PAL allows key objects of any type, PAL can search a dictionary for numeric and name key objects more effeciently than any other type of object. Therefore, the programmer should seriously consider using only numeric or name objects as key entries within a dictionary.

In order to take advantage of the more efficient name object searching, PAL automatically converts string objects specified as keys to name objects. PAL does not convert string objects specified as value entries within a dictionary. As a result, the following two dictionary definitions result in exactly the same dictionary within the printer's memory.

```
<<123 (1stValue)   /2ndKey /2ndData   45.76 /3rdData   (4thKey) 95.11>>
<<123 (1stValue)   /2ndKey /2ndData   45.76 /3rdData   /4thKey 95.11>>
```

The following table shows the organization of this dictionary within the printer's memory.

Key	Value
123	(1stValue)
/2ndKey	/2ndData
45.76	/3rdData
/4thKey	95.11

3.2.3. Procedure Objects

Procedure objects contain a series of objects which the programmer can instruct PAL to execute at a future time. PAL provides the "{" and "}" operators for defining procedures. These operators do not work like the "[," "]," "<<," and ">>" operators. Instead, the "{" operator instructs PAL to begin recording the following PAL operations and objects into a procedure object. The "}" operator instructs PAL to stop recording.

PAL also allows nested procedure definitions. This means that one procedure definition, enclosed in the "{" and "}" operators, can contain another procedure, enclosed in its own set of "{" and "}" operators.

While PAL records the operators and objects contained within the "{" and "}" operators, PAL does not perform any of the operations for the operators it encounters. One exception to this rule exists. PAL does continue to substitute immediately evaluated names with their associated objects.

Once PAL encounters the closing "}" operator and stops recording the procedure, PAL places the procedure object on the top of the operand stack. PAL does not attempt to execute the procedure at that time.

The programmer can treat a procedure object in many of the same ways as the programmer can treat array or dictionary objects. This includes the ability to store procedure objects within other composite objects. Except for allowing the programmer to execute the procedure when desired, PAL treats procedures as any other data objects.

As an example of this flexibility, consider a case where the programmer wishes to print thermal labels for various different parts within a company's inventory. Each part requires a label with different information and a different overall layout.

The programmer could create, within the printer's memory, a dictionary containing all the part numbers for each of the parts in the company's inventory. The programmer could then associated a procedure with each of these part numbers within the dictionary.

When the programmer wishes to print a label for a particular part, the programmer need only tell PAL which dictionary and part number to use and PAL will recall the procedure for printing that label from the dictionary.

If the process requires additional information about the part, the dictionary could contain array objects associated with each part number rather than procedures. These arrays could contain all the information related to the part as well as the procedure for printing the part's label.

3.3. Internal Objects

Certain operations cause PAL to mix internal object types with the objects created by the programmer. The internal classification of object types includes the following.

Intrinsic Operator
File
Font

3.3.1. Intrinsic Operator Objects

Intrinsic operator objects actually instruct the PAL interpreter to perform one of the numerous operations supported by PAL. The PAL interpreter contains a special dictionary, called systemdict, which associates name objects with intrinsic operator objects.

When PAL encounters an executable name, PAL searches for the name in systemdict. When PAL locates the name, it recovers the object associated with the name in the dictionary. In most cases, PAL will find an intrinsic operator object associated with the name. PAL then performs the action indicated by the instrinsic operator.

Therefore, systemdict establishes the association between a particular name and an intrinsic operation. PAL allows the programmer to supercede the associations in this dictionary. As a result, unlike other programming languages, PAL does not really treat the default names associated with each operation as reserved words. However, changing the definition of PAL's standard names only serves to make the programmer's PAL code harder to understand.

3.3.2. File Objects

The PAL language supports the concept of a data file. However, the location and naming of data files can vary from one PAL printer to the next. Typically, files may reside either in the printer's flash memory or in flash provided in the optional RTC card..

A file can also represent some input/output device available on the printer. As an example, the programmer can create a PAL program which reads data from the same host interface supplying commands to the PAL interpreter.

In order to keep track of the various information related to accessing of files, PAL creates a file object when the programmer opens a file. PAL then places this file object on the operand stack. The programmer can then save this object in order to access the file at a later time. Whenever the programmer wishes to access the file, the programmer places the file object back onto the stack and sends PAL the operator associated with the desired file access.

The file object references data private to the PAL interpreter. The interpreter does not allow the PAL programmer direct access to this information. However, PAL does provide operators which allow the programmer to indirectly access some of the file object information.

3.3.3. Font Objects

Each PAL printer contains a set of predefined fonts for drawing characters. Each font has a dictionary which defines all of the characteristics of that font.

Normally, a PAL programmer can view this dictionary as the font itself. The PAL operators which work with fonts accept this dictionary as an indication of which font to manipulate.

A font dictionary has the exact same structure as any other PAL dictionary. Therefore, the programmer may freely access the entries within any font dictionary. However, only the most experienced PAL programmers should even consider altering the contents of a font dictionary.

4. Operators

This section uses a consistent set of notation rules to summarize the operation of the numerous operators available under the PAL language. The operator usage summary lines show the operator written in a **`monospaced bold font`**. For easier reading, this manual also uses a sans-serif upright font for operators listed within the main text.

The list of objects which the operator expects to find on the operand stack appear to the left of the operator. The list of objects which the operator leaves on the operand stack appear to the right of the operator.

The text refers to the objects which the operator expects to find on the stack as the operator's *parameters*. The text refers to the objects which the operator leaves on the stack as the operator's *results*.

The operator usage summary lines use *tokens* to represent the positions of each parameter or result object. The text shows tokens in a *sans-serif italic font*.

The usage summary lines may enclose some parameters in square brackets — "[" and "]". Square brackets enclose optional parameters which the programmer may omit when not required. Usage summary lines may also use an abbreviated ellipse ("..") to indicate a range of parameters.

The name of every token shown includes a suffix which indicates the object type for the parameter or result. The following table lists the suffixes and their associated object types.

Suffix	Object Type
Any	Any
Array	Array
Bool	Boolean
Dict	Dictionary
File	File
Int	Integer
Mark	Mark
Name	Name
Null	Null
Num	Fixed-Point or Integer
Proc	Procedure
Str	String
Text	Name or String

4.1. Alphabetical Summary

Any	**==**	
	<<...>>	*Dict*
	[...]	*Array*
AnyNum	**abs**	*AbsNum*
Any1Num Any2Num	**add**	*SumNum*
Any1Bool Any2Bool	**and**	*AndBool*
Any1Int Any2I|nt	**and**	*AndInt*
ElementsInt	**array**	*NullArray*
DataAny [*CtrlDict*] *FormatName*	**_barcode**	
AnyDict	**begin**	
AnyProc	**bind**	*BoundProc*
AnyInt ShiftInt	**bitshift**	*ShiftedInt*
AnyNum	**ceiling**	*CeilingNum*
NAny..1Any	**clear**	
	cleartomark	
	closepath	
LeadStr TrailStr	**concat**	*ConcatStr*
NAny..1Any NInt	**copy**	*NAny..1Any NAny..1Any*
1Array 2Array	**copy**	*2Array*
1Dict 2Dict	**copy**	*2Dict*
1Str 2Str	**copy**	*2Str*
NAny..1Any	**count**	*NAny..1Any NInt*
Mark NAny..1Any	**counttomark**	*Mark NAny..1Any NInt*
	currentdict	*CurDict*
	currentgray	*LevelFxp*
	currentpoint	*XNum YNum*
ValNum DummyStr	**cvs**	*DecStr*
LiteralFile	**cvx**	*ExecFile*
LiteralName	**cvx**	*ExecName*
KeyName DataAny	**def**	
NameText FontDict	**definefont**	*FontDict*
FileStr AccessStr	**_devicefile**	*OpenFile*
PairsInt	**dict**	*EmptyDict*
DividendNum DivisorNum	**div**	*QuotientFxp*
BBoxArray	**dspclear**	
ColumnNum LineNum	**dspmovecursor**	
ColumnNum LineNum	**dspmoveto**	
ControlDict	**dspsetcursor**	
AnyStr	**dspstring**	
Any	**dup**	*Any Any*
	end	
1Any 2Any	**eq**	*Bool*
	erasepage	
1Any 2Any	**exch**	*2Any 1Any*
Any	**exec**	
	execexit	
FormDict	**execform**	
	executive	
	exit	
FileStr AccessStr	**file**	*OpenFile*
OpenFile	**fileposition**	*PositionInt*
FontName	**findfont**	*FontDict*
AnyNum	**floor**	*FloorNum*
BgnNum IncNum EndNum AnyProc	**for**	
Any1Num Any2Num	**ge**	*Bool*
Any1Text Any2Text	**ge**	*Bool*
AnyArray IndexInt	**get**	*ElementAny*
AnyDict KeyAny	**get**	*ValueAny*
AnyStr IndexInt	**get**	*CharInt*
AnyArray IndexInt LengthInt	**getinterval**	*SubArray*

AnyStr IndexInt LengthInt	**getinterval**	*SubStr*
	globaldict	*GlobalDict*
Any1Num Any2Num	**gt**	*Bool*
Any1Text Any2Text	**gt**	*Bool*
DividendInt DivisorInt	**idiv**	*QuotientInt*
AnyBool TrueProc	**if**	
AnyBool TrueProc FalseProc	**ifelse**	
WNum HNum PolBool TmArray SrcProc	**imagemask**	
Any1Bool Any2Bool	**_imp**	*ImpBool*
Any1Int Any2Int	**_imp**	*ImpInt*
NAny..0Any IndexInt	**index**	*NAny..0Any IndexedAny*
	initgraphics	
	initmatrix	
AnyDict KeyAny	**known**	*Bool*
Any1Num Any2Num	**le**	*Bool*
Any1Text Any2Text	**le**	*Bool*
AnyArray	**length**	*ElementsInt*
AnyDict	**length**	*PairsInt*
AnyStr	**length**	*CharsInt*
XNum YNum	**lineto**	
	_localtime	*TimeArray*
AnyProc	**loop**	
Any1Num Any2Num	**lt**	*Bool*
Any1Text Any2Text	**lt**	*Bool*
AnyStr SetStr	**_ltrim**	*TrimmedStr*
AnyFontDict TmArray	**makefont**	*TmFontDict*
	mark	*mark*
LimitsArray PagesInt	**_measurepage**	*SizeArray*
DividendInt DivisorInt	**mod**	*RemainderInt*
XNum YNum	**moveto**	
Any1Num Any2Num	**mul**	*ProductNum*
1Any 2Any	**ne**	*Bool*
AnyNum	**neg**	*NegNum*
	newpath	
AnyBool	**not**	*NotBool*
AnyNum	**not**	*NotNum*
	null	*Null*
Any1Bool Any2Bool	**or**	*OrBool*
Any1Int Any2Int	**or**	*OrInt*
ScoreSrr	**_play**	
Any	**pop**	
AnyStr	**print**	
AnyArray IndexInt ElementAny	**put**	
AnyDict KeyAny ValueAny	**put**	
AnyStr IndexInt CharInt	**put**	
TargetArray IndexInt SourceArray	**put**	
TargetStr IndexInt SourceStr	**put**	
AnyArray IndexInt SubArray	**putinterval**	
AnyStr IndexInt SubStr	**putinterval**	
	quit	
OpenFile AnyStr	**readstring**	*ReadStr*
CountInt AnyProc	**repeat**	
XDeltaNum YDeltaNum	**rlineto**	
XDeltaNum YDeltaNum	**rmoveto**	
AngleNum	**rotate**	
AnyNum	**round**	*RoundedNum*
AnyStr SetStr	**_rtrim**	*TrimmedStr*
XScaleNum YScaleNum	**scale**	
AnyFontDict ScaleNum	**scalefont**	*ScaledFontDict*
AnyStr SearchStr	**search**	*PostStr MatchStr PreStr* **true**
AnyStr SearchStr	**search**	*AnyStr* **false**
OpenFile PositionInt	**setfileposition**	
ScaledFontDict	**setfont**	
LevelNum	**setgray**	
CapInt	**setlinecap**	

WidthNum	**setlinewidth**	
TimeArray	**_setlocaltime**	
ControlDict	**setpagedevice**	
ShowStr	**show**	
	showpage	
PagesNum	**_showpages**	
CharsInt	**string**	*NullStr*
AnyStr	**stringwidth**	*XDeltaNum YDeltaNum*
	stroke	
Any1Num Any2Num	**sub**	*DifNum*
XTransNum YTransNum	**translate**	
	trap	
AnyNum	**truncate**	*TruncatedNum*
AnyDict KeyAny	**undef**	
	userdict	*userdict*
	vmstatus	*BytesInt*
OpenFile AnyStr	**writestring**	
Any1Bool Any2Bool	**xor**	*XorBool*
Any1Int Any2Int	**xor**	*XorInt*

==

Description

Writes the PAL language format of any object to %stdout.

Usage

Any ==

Any Any object type. Object which interpreter will write to %stdout.

Comments

If the stack contains a composite object, the interpreter will also write all objects which comprise the composite object. Writing of composite objects will continue through all nesting levels.

Some objects which can reside in memory do not have corresponding PAL language representations. For example, the programmer cannot include intrinsic operator objects and file objects directly within PAL source code.

Initially, intrinsic operator objects only exist within the system dictionary. The programmer accesses these intrinsic operator objects by referencing them using the key names associated with them in the system dictionary. File objects only result from the opening of a file.

In cases where a object does not have a PAL language representation, the interpreter writes the object using two hyphens ("--") before and after a text description of the object. For intrinsic operator objects, the interpreter uses the same text as the name associated with the object in the system dictionary. For example, the system dictionary contains the literal name add associated with the intrinsic operator object for the add operation. Therefore, the interpreter writes the intrinsic operator object for the add operation as `"--add--"`.

The interpreter provides the == operator primarily for debugging purposes. However, the operator can also prove useful for uploading data to a host computer provided the host system programs can accept the data in PAL format.

<<...>>

Description

Operator pair used to define a dictionary data object.

Usage

<<...>> Dict

Dict Dictionary. Object defined by operator pair.

Comments

In other programming languages, the language would treat symbols like these as syntactical in nature as opposed to executable. However, under PAL, the PAL interpreter executes these symbols in the same manner as add or any other PAL operator.

The PAL interpreter executes the opening ("**<<**") and closing ("**>>**") symbols as completely independent operators. The opening operator does nothing more than push a mark object onto the operand stack. The closing operator instructs the PAL interpreter to build a dictionary object from all objects on the top of the operand stack down to, but not including, the top most mark object. After PAL removes all the objects from the stack and places them into the new dictionary, it discards the mark object from the top of the stack.

The programmer must supply pairs of objects between the opening and closing symbols. In other words, an even number of objects should appear between the "**<<**" and "**>>**" symbols.

The PAL interpreter uses the first object of each pair as the key under which to store the second object within the dictionary. For example, the following sequence creates a dictionary with two entries. Each entry consists of a key and an associated value.

```
<< /Key1 (data1) 12 [78 95] >>
```

The first dictionary entry created will contain the string (data1) under the literal name key /Key1. The second entry created will contain the array of two numbers [78 95] under the numeric key 12.

The reader should note that PAL dictionaries may contain mixed types of keys and values. This provides maximum flexibility for the PAL programmer. However, certain data types make for more efficient search keys than other data types. For example, when later instructed to do so, the PAL interpreter can locate a literal name or numeric key entry faster than an array or other type of key.

Since the interpreter can locate a literal name faster than a string, the interpreter automatically converts strings used as keys to literal names before storing them into the dictionary. The interpreter only performs this conversion for key entries within the dictionary. The interpreter does not convert a string placed into a dictionary as a value object associated with a key object. For example, the following two dictionary definitions will result in exactly the same dictionary once the interpreter converts the key strings in the first definition.

```
<< (Key1) (data1) (Key2) /data2 >>
<< /Key1 (data1) /Key2 /data2 >>
```

Once the interpreter has created the dictionary, it pushes the dictionary object onto the top of the operand stack.

Hints

PAL dictionaries function in the same manner as key indexed data files. Using a dictionary, a programmer can create a file of key accessed data records directly within the PAL printer's memory. By associating an array with each key entry within the dictionary, the programmer can recall an entire record of data simply by supplying the key associated with the array. The array can contain all the data items for each record.

[...]

Description

Operator pair used to define an array object.

Usage

[*...***]** *Array*

Array Array. Object defined by operator pair.

Comments

In other programming languages, the language would treat symbols like these as syntactical in nature as opposed to executable. However, under PAL, the PAL interpreter executes these symbols in the same manner as add or any other PAL operator.

The PAL interpreter executes the opening ("[") and closing ("]") symbols as completely independent operators. The opening operator does nothing more than push a mark object onto the operand stack. The closing operator instructs the PAL interpreter to build an array object from all objects on the top of the operand stack down to, but not including, the top most mark object. After PAL removes all the objects from the stack and places them into the new array, it discards the mark object from the top of the stack.

Unlike many other programming languages, PAL arrays may contain mixed data types. These data types can include dictionaries, other arrays, and other composite data types. The following example creates an array containing four entries.

```
[(hello) [1 2 3] <</MyKey (me) /YourKey (you)>> /LitName]
```

The example specifies the string (hello) as the first array entry, the three element array [1 2 3] as the second entry, the two entry dictionary <</MyKey (me) /YourKey (you)>> as the third entry, and the literal name /LitName as the fourth entry.

Once the interpreter has created the array, it pushes the array object onto the top of the operand stack.

Hints

PAL arrays can function in the same manner as random access data files. Using an array, a programmer can create a file of data records directly within the PAL printer's memory. By creating an array containing arrays, the programmer can recall an entire record of data simply by supplying the index of the minor array record within the major array file. The minor array record can contain all the data items for each record.

abs

Description

Returns the absolute value of any number.

Usage

AnyNum **abs** *AbsNum*

AnyNum	Integer or fixed-point. Number from which to return absolute value.
AbsNum	Integer or fixed-point. Absolute value of *AnyNum*. Same object type as *AnyNum*.

Comments

The abs operator pops the top object from operand stack, calculates the object's absolute value, and pushes the result onto the operand stack. The result's type will match the original value's type.

add

Description

Adds two numbers and returns the sum.

Usage

Any1Num Any2Num **add** *SumNum*

Any1Num	Integer or fixed-point. First number to add.
Any2Num	Integer or fixed-point. Second number to add.
SumNum	Integer or fixed-point. Integer if *Any1Num* and *Any2Num* are both integer, otherwise fixed-point. Sum of *Any1Num* and *Any2Num*.

Comments

The add operator pops the top two objects from operand stack, adds them together, and pushes the result back onto the operand stack. The interpreter must find two numeric objects on the top of the stack or a typecheck error will result.

If the stack contains two integer objects, the interpreter will perform integer addition and push an integer result onto the stack. The interpreter will perform fixed point addition and push a fixed-point result if the stack contains a fixed-point object as either operand.

and

Description

Performs a logical or bit-wise *and* operation on two boolean or integer values.

Usage

Any1Bool Any2Bool **and** *AndBool*
Any1Int Any2Int **and** *AndInt*

Any1Bool Boolean. First operand for the logical *and* operation.

Any2Bool Boolean. Second operator for the logical *and* operation.

AndBool Boolean. Result of the logical *and* operation.

Any1Int Integer. First operand for the bit-wise *and* operation.

Any2Int Integer. Second operand for the bit-wise *and* operation.

AndInt Integer. Result of the bit-wise *and* operation.

Comments

The following table lists the results of performing the logical *and* operation on two boolean values.

		Any1Bool	
		false	true
Any2Bool	false	false	false
	true	false	true

The following table lists the results for each bit position when performing the bit-wise *and* operation on two integer values.

		Any1Int	
		0	1
Any2Int	0	0	0
	1	0	1

array

Description

Creates an array entirely consisting of null objects.

Usage

ElementsInt **array** *NullArray*

ElementsInt Integer. Number of elements to include within array.

NullArray Array. Array containing *ElementsInt* null objects.

Comments

The array and [...] operators perform similar functions. However, the [...] operators require the programmer to supply initialization data for a new array. The array operator does not require initialization data. The array operator automatically initializes the array with null objects. This provides a simplified means for programmers to create arrays which will receive their data at a later time.

_barcode

Description

Draws a bar code in a specified format.

Usage

DataStr [*CtrlDict*] *FormatName* **_barcode**

DataStr String. This is the data to be encoded in the bar code. See bar code specific discussions below for data requirements for each bar code format. See appendix A., *Bar Code Considerations* for a description of the capabilities, limitations, and special rules for PAL bar codes.

CtrlDict Optional dictionary. Contains entries which provide additional control over the formatting of the bar code drawn. The usage of each entry varies with the bar code format. See comments and format specific discussions for additional details.

Each dictionary entry has a full name and an abbreviation. Some also have aliases and abbreviations for the aliases. These abbreviations and aliases are provided to simplify the programming task. All variations on an entry name may be used interchangeably.

FormatName Literal name. Name of bar code format to draw. See discussions of each format for specific names.

Comments

The *DataStr* operand provides the data from which the PAL interpreter will build the bar code pattern. The contents of the string depends upon the bar code format specified by the *FormatName* parameter. Some formats may accept data objects of more than one type. The text below discusses the individual bar code formats.

The *FormatName* parameter specifies the format of the bar code to draw. The _barcode operator requires a literal name for the *FormatName* parameter. The text below gives a list of the bar code formats supported by PAL. The number and combination of bar code formats on a given PAL printer may vary. The reader should check the documentation for each printer model to determine the bar code formats supported by the printer. See appendix A., *Bar Code Considerations* for a discussion of the problems that may be encountered when moving a PAL program between different PAL printer models.

_barcode: Code 128

DataStr [*CtrlDict*] **/Code128 _barcode**

This symbology conforms to the USS-128 specification.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/CheckDigit /CD	Boolean. true instructs PAL to calculate the check digit and insert it into the data string. false will instruct PAL to use the data string without modification. Default value = true.
/Height /H	Integer or fixed-point. Specifies the height of the bars. This value should be at least 0.25 inches (6.35 mm) or 15% of the bar code symbol length, whichever is greater. Note that this is the height of only the bars and does not include the human readable text, if any. The resulting overall bar code image may be taller than Height. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/HRAbove /HRA	Boolean. true instructs PAL to place the human readable text above the bar code. false instructs PAL to place the human readable text below the bar code. If HRShow is false, this entry is ignored. Default value = false.
/HRShow /HR	Boolean. true instructs PAL to automatically print the human readable text along with the bar code. false instructs PAL not to include the human readable. Default value = true.
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars drawn as part of the bar code. This is generally referred to as the X dimension. Default value = 0.72 (1/100" / 0.254 mm with default CTM).
/UCC128 /EAN128 /U128 /E128	Boolean. true instructs PAL to use UCC-128/EAN-128 format. false instructs PAL to use Code-128. Default setting = false.

Usage Notes

Code 128 has three different code sets designated Code A, Code B, Code C. These code sets provide different mappings of the 106 available Code 128 symbols to ASCII. These mappings, along with the PAL strings that generates the symbols, are shown in the table below. Code A provides most symbols, the capital letters, and ASCII control characters. Code B provides the complete ASCII symbol set, but no control characters. Code C provides a very dense packing of numeric strings. Two decimal digits are packed into one Code 128 symbol.

The generation of Code 128 bar codes is controlled by several special control characters included in the Code 128 set. PAL is instructed to generate these special characters with a two-character

string beginning with a tilde (~). To get the character tilde, two tildes (~~) are used. These characters are shown in the following table.

Three characters, \ (and), require special handling in PAL. The backslash character, \, is an escape character for encoding special characters in PAL strings. To get the backslash character itself, you must use a double backslash (\\). The parenthesis characters are used in PAL to indicate the beginning and ending of a string. To include these characters in a string, they must be proceeded by a backslash (*i.e.*, \(and \)).

Code C values are specified as a string of ASCII numerals. Since the Code 128 symbols are made up of pairs of numerals, there must be an even number of numerals in the string. If PAL encounters a non-numeral while building a Code C bar code, the string will be rejected with a rangecheck error.

For Code A and Code B, the character used for a Code 128 symbol is usually the matching ASCII character. Since there could be a problem getting some of the ASCII control characters in Code A through some communication systems (not to mention PAL itself), these Code A characters are requested by using the equivalent character from Code B (*e.g.* for ACK, use (f)). See the table below for a complete list of all the PAL strings needed to produce Code 128 symbols. Any inappropriate ASCII character in the data string will cause the string to be rejected with a rangecheck error.

A Code 128 bar code must start with a start code indicating the code to be used. If the data string does not begin with one of the three start codes (~a, ~b, or ~c), a start code for Code B is supplied by PAL. Code 128 allows the code to be changed within the bar code. If the code is changed, PAL automatically begins using the rules for the new code.

Value	Code A	A String	Code B	B String	Code C	C String
0	SP	( )	SP	( )	00	(00)
1	!	(!)	!	(!)	01	(01)
2	"	(")	"	(")	02	(02)
3	#	(#)	#	(#)	03	(03)
4	$	($)	$	($)	04	(04)
5	%	(%)	%	(%)	05	(05)
6	&	(&)	&	(&)	06	(06)
7	'	(')	'	(')	07	(07)
8	(	(\()	(	(\()	08	(08)
9	)	(\))	)	(\))	09	(09)
10	*	(*)	*	(*)	10	(10)
11	+	(+)	+	(+)	11	(11)
12	,	(,)	,	(,)	12	(12)
13	-	(-)	-	(-)	13	(13)
14	.	(.)	.	(.)	14	(14)
15	/	(/)	/	(/)	15	(15)
16	0	(0)	0	(0)	16	(16)
17	1	(1)	1	(1)	17	(17)
18	2	(2)	2	(2)	18	(18)
19	3	(3)	3	(3)	19	(19)
20	4	(4)	4	(4)	20	(20)
21	5	(5)	5	(5)	21	(21)
22	6	(6)	6	(6)	22	(22)
23	7	(7)	7	(7)	23	(23)
24	8	(8)	8	(8)	24	(24)
25	9	(9)	9	(9)	25	(25)
26	:	(:)	:	(:)	26	(26)
27	;	(;)	;	(;)	27	(27)
28	<	(<)	<	(<)	28	(28)
29	=	(=)	=	(=)	29	(29)
30	>	(>)	>	(>)	30	(30)
31	?	(?)	?	(?)	31	(31)
32	@	(@)	@	(@)	32	(32)
33	A	(A)	A	(A)	33	(33)
34	B	(B)	B	(B)	34	(34)
35	C	(C)	C	(C)	35	(35)
36	D	(D)	D	(D)	36	(36)
37	E	(E)	E	(E)	37	(37)
38	F	(F)	F	(F)	38	(38)
39	G	(G)	G	(G)	39	(39)
40	H	(H)	H	(H)	40	(40)
41	I	(I)	I	(I)	41	(41)
42	J	(J)	J	(J)	42	(42)
43	K	(K)	K	(K)	43	(43)
44	L	(L)	L	(L)	44	(44)
45	M	(M)	M	(M)	45	(45)
46	N	(N)	N	(N)	46	(46)
47	O	(O)	O	(O)	47	(47)
48	P	(P)	P	(P)	48	(48)
49	Q	(Q)	Q	(Q)	49	(49)
50	R	(R)	R	(R)	50	(50)
51	S	(S)	S	(S)	51	(51)
52	T	(T)	T	(T)	52	(52)
53	U	(U)	U	(U)	53	(53)
54	V	(V)	V	(V)	54	(54)
55	W	(W)	W	(W)	55	(55)
56	X	(X)	X	(X)	56	(56)
57	Y	(Y)	Y	(Y)	57	(57)
58	Z	(Z)	Z	(Z)	58	(58)
59	[	([)	[	([)	59	(59)
60	\	(\\)	\	(\\)	60	(60)
61	]	(])	]	(])	61	(61)
62	^	(^)	^	(^)	62	(62)
63	_	(_)	_	(_)	63	(63)
64	NUL	(`)	`	(`)	64	(64)
65	SOH	(a)	a	(a)	65	(65)
66	STX	(b)	b	(b)	66	(66)
67	ETX	(c)	c	(c)	67	(67)
68	EOT	(d)	d	(d)	68	(68)
69	ENQ	(e)	e	(e)	69	(69)
70	ACK	(f)	f	(f)	70	(70)
71	BEL	(g)	g	(g)	71	(71)
72	BS	(h)	h	(h)	72	(72)
73	HT	(i)	i	(i)	73	(73)
74	LF	(j)	j	(j)	74	(74)
75	VT	(k)	k	(k)	75	(75)
76	FF	(l)	l	(l)	76	(76)
77	CR	(m)	m	(m)	77	(77)
78	SO	(n)	n	(n)	78	(78)
79	SI	(o)	o	(o)	79	(79)
80	DLE	(p)	p	(p)	80	(80)
81	DC1	(q)	q	(q)	81	(81)
82	DC2	(r)	r	(r)	82	(82)
83	DC3	(s)	s	(s)	83	(83)
84	DC4	(t)	t	(t)	84	(84)
85	NAK	(u)	u	(u)	85	(85)
86	SYN	(v)	v	(v)	86	(86)
87	ETB	(w)	w	(w)	87	(87)
88	CAN	(x)	x	(x)	88	(88)
89	EM	(y)	y	(y)	89	(89)
90	SUB	(z)	z	(z)	90	(90)
91	ESC	({)	{	({)	91	(91)
92	FS	(|)	|	(|)	92	(92)
93	GS	(})	}	(})	93	(93)
94	RS	(~~)	~	(~~)	94	(94)
95	US	(DEL)	DEL	(DEL)	95	(95)
96	FNC3	(~3)	FNC3	(~3)	96	(96)
97	FNC2	(~2)	FNC2	(~2)	97	(97)
98	SHFT	(~S)	SHFT	(~S)	98	(98)
99	CodC	(~C)	CodC	(~C)	99	(99)
100	CodB	(~B)	FNC4	(~4)	CodB	(~B)
101	FNC4	(~4)	CodA	(~A)	CodA	(~A)
102	FNC1	(~1)	FNC1	(~1)	FNC1	(~1)
103	Start CODE A (~a)					
104	Start CODE B (~b)					
105	Start CODE C (~c)					

If the UCC128 or EAN128 flags are true, the symbol is to be used as part of the UCC or EAN system as a supplemental code. Both systems use the same format. A UCC-128 symbol is a standard Code 128 symbol that begins with one of the start characters immediately followed by an FNC1 character. FNC1 has been reserved to exclusively indicate a UCC 128/EAN 128 symbol. The remainder of the characters are fairly free-form. The first two characters are usually numeric and indicate the format of the remainder of the symbol. These remaining characters may be alphabetic and/or numeric characters and may be any length up to 30 characters. When PAL is requested to build a symbol in this format, it checks that the character after the start code is FNC1. If it is not, an FNC1 is inserted.

The generated image includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 0.5 inch tall Code 128 bar code with 0.01 inch narrow bars and no human readable. The data string begins with Code B, switches to Code C for the numbers, and ends up in Code A with an ASCII control character (ETX).

```
(~bCode 128 ~C12345678~Ac) <</HRShow false>> /Code128
_barcode
```

_barcode: Code 39

DataStr [*CtrlDict*] **/Code39 _barcode**

This symbology conforms to the USS-39 specification.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/CheckDigit /CD	Boolean. true instructs PAL to automatically calculate the check digit and insert it at the end of the data string just before the stop character. false instructs PAL to use the data string without modification. A computed check digit will also be displayed in the human readable text. Default value = false.
/Height /H	Integer or fixed-point. Specifies the height, in current user units, of the bar code. This value should be at least 0.25 inches (6.35 mm) or 15% of the bar code symbol length, whichever is greater. Note that this is the height of the bars and does not include the human readable text, if any. The resulting overall bar code image may be taller than Height. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/HRAbove /HRA	Boolean. true instructs PAL to print the human readable above the bar code. false instructs PAL to print the human readable below the bar code. If HRShow is false, this entry is ignored. Default value = false.
/HRShow /HR	Boolean. true instructs the PAL to draw the human readable text along with the bar code. false instructs PAL to not draw the human readable text. Default value = true.
/HRShowStartStop /SS	Boolean. true instructs PAL to include the start and stop characters (*) when drawing the human readable text. false instructs PAL to not include the start and stop characters as part of the human readable. Although the specification suggests these characters not to be printed, traditionally they are printed. For that reason, the default is set to print them. If HRShow is false, this entry is ignored. Default value = true.
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars drawn as part of the bar code. This is generally referred to as the X dimension. Default value = 0.72 (1/100" / 0.254 mm with default CTM).
/WideRatio /Ratio /WR /R	Integer or fixed-point. Specifies the ratio of the wide bars to the narrow bars. The value is a multiplier which PAL applies to the narrow bar width in order to establish the width of the wide bars. Default value = 3.0.

Character Set

Valid characters for Code 39 bar codes are the digits 0 through 9, the capital letters A through Z, a space, and the characters - . $ / + % and *. Any other characters will cause the string to be rejected with a rangecheck error.

Extended Character Set

Code 39 supports a scheme for encoding the full ASCII character set. By combining the characters $ + % and / with valid Code 39 characters as shown in the following table, all ASCII characters may be encoded.

ASCII	PAL String	ASCII	PAL String	ASCII	PAL String	ASCII	PAL String
NUL	(%U)	SP	( )	@	(%V)	`	(%W)
SOH	($A)	!	(/A)	A	(A)	a	(+A)
STX	($B)	"	(/B)	B	(B)	b	(+B)
ETX	($C)	#	(/C)	C	(C)	c	(+C)
EOT	($D)	$	(/D)	D	(D)	d	(+D)
ENQ	($E)	%	(/E)	E	(E)	e	(+E)
ACK	($F)	&	(/F)	F	(F)	f	(+F)
BEL	($G)	'	(/G)	G	(G)	g	(+G)
BS	($H)	(	(/H)	H	(H)	h	(+H)
HT	($I)	)	(/I)	I	(I)	i	(+I)
LF	($J)	*	(/J)	J	(J)	j	(+J)
VT	($K)	+	(/K)	K	(K)	k	(+K)
FF	($L)	,	(/L)	L	(L)	l	(+L)
CR	($M)	-	-	M	(M)	m	(+M)
SO	($N)	.	.	N	(N)	n	(+N)
SI	($O)	/	(/O)	O	(O)	o	(+O)
DLE	($P)	0	0	P	(P)	p	(+P)
DC1	($Q)	1	1	Q	(Q)	q	(+Q)
DC2	($R)	2	2	R	(R)	r	(+R)
DC3	($S)	3	3	S	(S)	s	(+S)
DC4	($T)	4	4	T	(T)	t	(+T)
NAK	($U)	5	5	U	(U)	u	(+U)
SYN	($V)	6	6	V	(V)	v	(+V)
ETB	($W)	7	7	W	(W)	w	(+W)
CAN	($X)	8	8	X	(X)	x	(+X)
EM	($Y)	9	9	Y	(Y)	y	(+Y)
SUB	($Z)	:	(/Z)	Z	(Z)	z	(+Z)
ESC	(%A)	;	(%F)	[	(%K)	{	(%P)
FS	(%B)	<	(%G)	\	(%L)	|	(%Q)
GS	(%C)	=	(%H)	]	(%M)	}	(%R)
RS	(%D)	>	(%I)	^	(%N)		(%S)
US	(%E)	?	(%J)	_	(%O)	DEL	(%T)

Usage Notes

The start and stop characters (*) need not be specified. If they are omitted, PAL will supply them.

The generated image includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 0.5 inch tall Code 39 bar code with 0.01 inch narrow bars and a ratio of 2.5:1. A check digit will be calculated and start/stop characters will be displayed.

```
(BAR CODE 39) <</WideRatio 2.5 /CheckDigit true>> /Code39
_barcode
```

_barcode: Code 93

DataStr [*CtrlDict*] **/Code93 _barcode**

This symbology conforms to the USS-93 specification.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/Height /H	Integer or fixed-point. Specifies the height, in current user coordinates, of the bar code. This value should be at least 0.25 inches (6.35 mm) or 15% of the bar code symbol length, whichever is greater. Note that this is the height of the bars and does not include the human readable text, if any. The resulting overall bar code image may be taller than Height. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/HRAbove /HRA	Boolean. true instructs PAL to draw the human readable text above the bar code. false instructs PAL to draw the human readable text below the bar code. If HRShow is false, this entry is ignored. Default value = false.
/HRShow /HR	Boolean. true instructs PAL to draw the human readable text along with the bar code. false instructs PAL to not draw the human readable text. Default value = true.
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars. This is generally referred to as the X dimension. Default value = 0.72 (1/100" / 0.254 mm with default CTM).

Character Set

Valid characters for Code 93 bar codes are the digits 0 through 9, the capital letters A through Z, a space, and the characters - . $ / + and %. Four special characters, [$] [%] [/] and [+], are also supported. To send these special characters, precede the character with ~. Thus, to specify [$], send ~$. Any other characters will cause the string to be rejected with a rangecheck error.

Extended Character Set

Code 93 supports a scheme for encoding the full ASCII character set. By combining special characters with valid Code 93 characters as shown in the following table, all ASCII characters may be encoded.

ASCII	PAL String	ASCII	PAL String	ASCII	PAL String	ASCII	PAL String
NUL	(~%U)	SP	( )	@	(~%V)	`	(~%W)
SOH	(~$A)	!	(~/A)	A	(A)	a	(~+A)
STX	(~$B)	"	(~/B)	B	(B)	b	(~+B)
ETX	(~$C)	#	(~/C)	C	(C)	c	(~+C)
EOT	(~$D)	$	$	D	(D)	d	(~+D)
ENQ	(~$E)	%	%	E	(E)	e	(~+E)
ACK	(~$F)	&	(~/F)	F	(F)	f	(~+F)
BEL	(~$G)	'	(~/G)	G	(G)	g	(~+G)
BS	(~$H)	(	(~/H)	H	(H)	h	(~+H)
HT	(~$I)	)	(~/I)	I	(I)	i	(~+I)
LF	(~$J)	*	(~/J)	J	(J)	j	(~+J)
VT	(~$K)	+	+	K	(K)	k	(~+K)
FF	(~$L)	,	(~/L)	L	(L)	l	(~+L)
CR	(~$M)	-	-	M	(M)	m	(~+M)
SO	(~$N)	.	.	N	(N)	n	(~+N)
SI	(~$O)	/	/	O	(O)	o	(~+O)
DLE	(~$P)	0	0	P	(P)	p	(~+P)
DC1	(~$Q)	1	1	Q	(Q)	q	(~+Q)
DC2	(~$R)	2	2	R	(R)	r	(~+R)
DC3	(~$S)	3	3	S	(S)	s	(~+S)
DC4	(~$T)	4	4	T	(T)	t	(~+T)
NAK	(~$U)	5	5	U	(U)	u	(~+U)
SYN	(~$V)	6	6	V	(V)	v	(~+V)
ETB	(~$W)	7	7	W	(W)	w	(~+W)
CAN	(~$X)	8	8	X	(X)	x	(~+X)
EM	(~$Y)	9	9	Y	(Y)	y	(~+Y)
SUB	(~$Z)	:	(~/Z)	Z	(Z)	z	(~+Z)
ESC	(~%A)	;	(~%F)	[	(~%K)	{	(~%P)
FS	(~%B)	<	(~%G)	\	(~%L)	|	(~%Q)
GS	(~%C)	=	(~%H)	]	(~%M)	}	(~%R)
RS	(~%D)	>	(~%I)	^	(~%N)	~	(~%S)
US	(~%E)	?	(~%J)	_	(~%O)	DEL	(~%T)

Usage Notes

Code 93 symbols have two check digits. The interpreter generates them automatically and they should not be included in the input string. The interpreter also automatically inserts start and stop characters.

The generated image includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 1.0 inch tall Code 93 bar code with 0.01 inch wide narrow bars.

```
(BAR CODE 93) <</H 72>> /Code93 _barcode
```

Codabar

DataStr [*CtrlDict*] **`/Codabar _barcode`**

This symbology conforms to the USS-Codabar specification. This specification is derived from traditional Codabar and is fully compatible with it in all applications.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/CheckDigit /CD	Boolean. No check digit is required for this symbology. If this entry is set to true, a check digit will be calculated using the algorithm suggested in the USS-Codabar specification. Default value = false.
/Height /H	Integer or fixed-point. Specifies the height, in current user coordinates, of the bar code. This value should be at least 0.25 inches (6.35 mm) or 15% of the bar code symbol length, whichever is greater. Note that this is the height of the bars and does not include the human readable text, if any. The resulting overall bar code image may be taller than Height. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/HRAbove /HRA	Boolean. true instructs PAL to draw the human readable text above the bar code. false instructs PAL to draw the text below the bar code. Default value = false. If HRShow is false, this entry is ignored.
/HRShow /HR	Boolean. true instructs PAL to draw the human readable text along with the bar code. false instructs PAL not to draw the human readable text. Default value = true.
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars. This is generally referred to as the X dimension. Default value = 0.72 (1/100" / 0.254 mm with default CTM).

Usage Notes

Valid characters for Codabar bar codes are the digits 0 through 9, the symbols - $: / . +, and the start/stop characters A B C or D. Any other characters will cause the string to be rejected with a rangecheck error.

Codabar symbols should start and end with one of the four start/stop characters. The use of start/stop characters varies with the application using the bar code. For this reason, PAL does not enforce the presence of these characters. If they are omitted, however, the resulting bar code will probably not scan.

The generated bit map includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 0.5 inch tall Codabar bar code with 0.01 inch narrow bars and the human readable text printed above the bar code.

```
(A12345678B) <</HRAbove true>> /Codabar _barcode
```

_barcode: EAN-8

DataStr [*CtrlDict*] **/EAN8 _barcode**

This symbology conforms to the General EAN specification.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/CheckDigit /CD	Boolean. true instructs PAL to automatically calculate the check digit and insert it at the end of the data string, just before the stop character. The check digit will also be part of the human readable text. false instructs PAL that the data string already contains a check digit and the string should be used without modification. Default value = false.
/Height /H	Integer or fixed-point. Specifies the height, in current user coordinates, of the bar code. For this bar code symbology, this value specifies the height of the entire bar code image including the human readable text. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars drawn as part of the bar code. This is generally referred to as the X dimension. Default value = 0.936 (0.013" / 0.33 mm with default CTM).

Usage Notes

Valid characters for EAN-8 bar codes are the digits 0 through 9. Any other characters will cause the string to be rejected with a rangecheck error.

EAN-8 symbols require exactly 8 digits. If fewer digits are provided, the string will be padded with leading zeros. If a check digit has been requested PAL computes the correct value and replaces the eighth digit with the new check digit. The data string should contain a dummy character in the eighth digit to receive the check digit. Specifying short strings causes the string to be padded with zeros and the last character being replaced by the check digit.

The generated image includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 1.13 inch tall EAN-8 bar code with 0.113 inch narrow bars. A check digit will be calculated.

```
(01234560) <</Height 72 1.13 mul /CheckDigit true>> /EAN8
_barcode
```

_barcode: EAN-13

DataStr [*CtrlDict*] **/EAN13 _barcode**

This symbology conforms to the General EAN specifications.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/AddOn2 /AO2	Boolean. true instructs PAL to add a 2-digit add-on bar code to the base bar code. false instructs PAL not to add the add-on bar code. Default value = false.
/AddOn5 /AO5	Boolean. true instructs PAL to add a 5 digit add-on bar code to the base bar code. false instructs PAL not to add the add-on bar code. Default value = false.
/CheckDigit /CD	Boolean. true instructs PAL to automatically calculate the check digit and insert it into the data string immediately before the stop character. The check digit will also appear as part of the human readable. false instructs PAL that the data strings already contains the check digit and to use the data string without modification. Default value = false.
/Height /H	Integer or fixed-point. Specifies the height, in current user coordinates, of the bar code. For this bar code symbology, the height includes the human readable text. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars drawn as part of the bar code. This is generally referred to as the X dimension. Default value = 0.936 (0.013" / 0.33 mm with default CTM).
/RandWt4 /RW4	Boolean. true instruct PAL to automatically calculate the internal check digit for a four-digit random weight (or price) field. false instructs PAL to not perform this calculation. Default value = false.
/RandWt5 /RW5	Boolean. true instructs PAL to automatically calculate the internal check digit for a five-digit random weight (or price) field. Default value = false.

Usage Notes

Valid characters for EAN-13 bar codes are the digits 0 through 9. Any other characters will cause the string to be rejected with a rangecheck error.

EAN-13 symbols require exactly 13 digits. If fewer digits are provided, the string will be padded with leading zeros. If a check digit has been requested, PAL computes the correct value and replaces the 13th digit with the new check digit. The data string should contain a dummy character in

the 13th digit to receive the check digit. Specifying short strings causes the string to be padded with zeros and the last character to be replaced by the check digit.

If the RandWt4 or the RandWt5 flags are true, the last 4 or 5 digits before the check digit contain a random weight or a price. This allows in-store marking of such items as meat or cheese. A check digit for the random weight is calculated and replaces the character immediately preceding the 4- or 5-digit field. Since a new check digit must be calculated for the entire, symbol, the CheckDigit flag should always be set to true when using random weight symbols.

If the AddOn2 or the AddOn5 flags are true, a 2- or 5-digit add-on bar code is added to the right of the EAN-13 symbol. Since the check digits for these add-on bar codes are implicit in the encoding, no check digit is specified for the add-on portion. To specify the text for a bar code with a supplement, simply add 2 or 5 digits to the end of a standard 13-digit EAN-13 number. (Don't forget that the character before the supplement will contain the normal check digit.)

The generated image includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 1.0 inch tall EAN-13 bar code with 0.013 inch narrow bars. A check digit will be calculated.

```
(0123456789010) <</Height 72  /CheckDigit true>> /EAN13
_barcode
```

The following example generates 1.13 inch tall EAN-13 bar code with 0.013 inch narrow bars and a 4-digit random weight field. Both zeros in the symbol will be replaced by check digits.

```
(212345012340) <</Height 72 1.13 mul /RandWt4 true
/CheckDigit true>> /EAN13 _barcode
```

The following example generates 1.13 inch tall EAN-13 bar code with 0.013 inch narrow bars and a 5-digit add-on bar code. Note that the 8 in the 13th position of the symbol is the check digit check digit.

```
(978078211054890000) <</Height 72 1.13 mul /AddOn5 true>>
/EAN13 _barcode
```

_barcode: Interleave 2 of 5

DataStr [*CtrlDict*] **/I2of5 _barcode**

This symbology conforms to the USS-I 2/5 specification.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/CheckDigit /CD	Boolean. true instructs PAL to automatically calculate a check digit based on the algorithm suggested in the USS-I2/5 specification. false instructs PAL to not calculate the check digit. Default value = false.
/Height /H	Integer or fixed-point. Specifies the height, in current user coordinates, of the bar code. This is the height of the bar codes bars and does not include the human readable text, if any. The resulting overall bar code image may be taller than Height. This value should be at least 0.25 inches (6.35 mm) or 15% of the bar code symbol length, whichever is greater. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/HRAbove /HRA	Boolean. true instructs PAL to draw the human readable text above the bar code. false instructs PAL to draw the human readable text below the bar code. If HRShow is false, this entry is ignored. Default value = false.
/HRShow /HR	Boolean. true instructs PAL to draw the human readable text along with the bar code. false instructs PAL to not draw the human readable text. Default value = true.
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user unts, of the narrow bars. This is generally referred to as the X dimension. Default value = 0.72 (1/100" / 0.254 mm with default CTM).
/WideRatio /Ratio /WR /R	Integer or fixed-point. Specifies the ratio of the wide bars to the narrow bars. The value specifies a multiplier which PAL applies to the narrow bar width in order to establish the width of the wide bars. Default value = 3.0.

Usage Notes

Valid characters for Interleaved 2-of-5 bar codes are the digits 0 through 9. Any other characters will cause the string to be rejected with a rangecheck error.

Interleaved 2-of-5 gets its name from the fact that two digits are encoded in one bar code character, one in the bars and one in the spaces. For this reason, digits must always be specified in pairs. If an odd number of digits is specified, PAL furnishes a leading zero. Thus, the string 123 would be encoded as 01 23.

The generated bit map includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 0.5 inch tall Interleaved 2-of-5 bar code with 0.01 inch narrow bars and a ratio of 2.5:1.

```
(12345678) <</WideRatio 2.5>> /I2of5 _barcode
```

_barcode: PDF-417

DataStr [*CtrlDict*] **/PDF417 _barcode**

This symbology conforms to the UPC Symbol Specification Manual.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/AddresseeId
/AI
String. Range 1..200 characters. Specifies the data for the PDF-417 "Addressee ID" macro field. The bar code will not include the Addressee ID macro field if this parameter is not specified or an empty string is specified. Default = Empty string (field not included in bar code).

/Aspect
/A
String. Specifies the height:width (height-to-width) aspect ratio of the overall two dimension bar code. The string has the format "(*Height*:*Width*)". *Height* specifies the height component of the ratio, and *Width* specifies the width component. For example, "/Aspect (2:1)" specifies a bar code which is twice as height as it is wide. Row and Column with override this dictionary entry. Default = (1:2).

/BlockCount
/BC
Boolean. true instructs PAL to include the PDF-417 "Block Count" macro field within the bar code. false instructs PAL to not include the Block Count macro field as part of the bar code. Default value = false.

/CheckSum
/CS
Boolean. true instructs PAL to include the PDF-417 "Checksum" macro field within the bar code. false instructs PAL to not include the field. Default = false.

/Cols
/C
Integer. Range 0..30. Establishes either the absolute or maximum number of data columns which comprise the bar code. The SizeFixed parameter controls the selection between absolute or maximum. Specifying Cols overrides the Aspect parameter. Specifying Cols without Rows results in PAL generating a bar code with the minimum number of rows based on the specified number of columns. Default = Cols not specified.

/EccPercent
/EP
Integer. Range 0..400. Establishes the amount of error detection and correction codes added to the user's data as a percentage of the amount of user data. This setting overrides the EccLevel setting unless the user explicitly specifies "/EccPercent 0". Default value = 10.

/EccLevel
/EL
Integer. Range 0..7. Establishes the error detections and correction security level as per the PDF-417 bar code specification. The EccPercent setting will override this setting unless the user explicitly specifies "/EccPercent 0". Default value = 0.

/FileId
/FI

String. Range 1..50 characters. Specifies the data for the PDF-417 “File ID” macro field. The bar code will not include the File ID macro field if this parameter is not specified or an empty string is specified. Default = Empty string (field not included in bar code).

/FileName
/FN

String. Range 1..200 characters. Specifies the data for the PDF-417 “File Name” macro field. The bar code will not include the File Name macro field if this parameter is not specified or an empty string is specified. Default = Empty string (field not included in bar code).

/FileSize
/FS

Boolean. true instructs PAL to include the PDF-417 “File Size” macro field within the bar code. false instructs PAL to not include the field. Default = false.

/Height
/H

Integer or fixed-point. Specifies the height, in current user units, of the bars in each row of the two dimensional bar code. Default value = 3 (0.5" / 12.7 mm with default CTM).

/NarrowWidth
/XWidth
/NW
/X

Integer or fixed-point. Specifies the width, in current user units, of the narrow bars drawn as part of the bar code. This is generally referred to as the X dimension. Default value = 0.72 points (1/100” / 0.254 mm with default CTM).

/Rows
/R

Integer. Range 0..90. Establishes either the absolute or maximum number of data rows which comprise the bar code. The SizeFixed parameter controls the selection between absolute or maximum. Specifying Rows overrides the Aspect parameter. Specifying Rows without Cols results in PAL generating a bar code with the minimum number of columns based on the specified number of rows. Default = Rows not specified.

/SenderId
/SI

String. Range 1..200 characters. Specifies the data for the PDF-417 “Sender ID” macro field. The bar code will not include the Sender ID macro field if this parameter is not specified or an empty string is specified. Default = Empty string (field not included in bar code).

/SizeFixed
/SF

Boolean. true instructs PAL to treat the Rows and Cols values as specifying the required size of the two dimension bar code. This will force the bar code to be the specified size. false instructs PAL to treat the Rows and Cols values as specifying only maximum sizes for the respective dimensions of the two dimensional bar code. Default = false.

/TimeStamp
/T

Integer. Specifies a value for the PDF-417 “Time Stamp” macro field. A value of -1 instructs PAL to automatically generate the Time Stamp field value from the printer’s internal real-time clock. Specifying -1 is only valid on PAL printers which include internal real-time clock services. Default = Time Stamp field not included in bar code.

/Truncate /TR	Boolean. true instructs PAL to not draw the right side indicators and right stop pattern for the bar code. false instructs PAL to draw these patterns. Default = false.

Usage Notes

Error correction coding (ECC) consists of additional data added to the user's data in order to facilitate recovery of the user's data even when the bar code has been damaged and cannot be fully scanned. PDF-417 provides two different mechanisms for specifying the amount of ECC to include within the bar code.

EccPercent allows the user to specify the amount of error correction codes to add to the data as a percentage of the data. The larger the EccPercent value, the more damage the bar code can sustain while still allowing the scanner to recover 100% of the user's data.

EccLevel allows the user to select the amount of error correction codes added to the data based on predetermined values established for PDF-417 bar codes. EccLevel setting manages error correction and detection during scanning based on the following equation.

$$faults = errors + 2 \times misdecodes$$

where

errors = the number of unscannable codewords.

misdecodes = number of misdecoded codewords.

The following table gives the number of *faults* per EccLevel setting which the scanning process can tolerate and still be able to recover 100% of the user's data.

EccLevel	*faults*
0	0
1	2
2	6
3	14
4	30
5	62
6	126
7	254
8	510

Due to the advanced nature of the two dimensional PDF-417 symbology, the control parameters associated with thesymbology have a much higher level of complexity.than more traditional one dimensional bar codes. Most of the configuration parameters provided by PAL exist to allow the user extra control over the creation of the bar code.

The user will find that allowing the printer to produce PDF-417 bar codes using the printer's built-in default parameters will prove as simple as printing single dimension bar codes. Therefore, users not familiar with details of PDF-417 bar codes should start by using the printer's built-in defaults. Then, if necessary, the user should add only one or two PDF-417 configuration parameters at a time in order to observe the affect of the parameters upon the bar code.

Users already famility with the details of PDF-417 bar codes will also already be familiar with the various configuration parameters provided by PAL.

_barcode: UPC-A

DataStr [*CtrlDict*] **/UPCA _barcode**

This symbology conforms to the UPC Symbol Specification Manual.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/AddOn2 /AO2	Boolean. true instructs PAL to add a two-digit add-on bar code to the base bar code. false instructs PAL not to add the add-on bar code. Default value = false.
/AddOn5 /AO5	Boolean. true instructs PAL to add a five-digit add-on bar code to the base bar code. false instructs PAL not to add the add-on bar code. Default value = false.
/CheckDigit /CD	Boolean. true instructs PAL to automatically calculate a check digit and insert it at the end of the data string, just before the stop character. The check digit will also be drawn as part of the human readable. false instructs PAL that the string already contains the check digit and that the string should be used without modification. Default value = false.
/Height /H	Integer or fixed-point. Specifies the height, in current user units, of the entire bar code image. For this bar code symbology, this is the height of the entire bar code image include the human readable text. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars drawn as part of the bar code. This is generally referred to as the X dimension. Default value = 0.936 (0.013" with default CTM).
/RandWt4 /RW4	Boolean. true instructs PAL to automatically compute the internal check digit for a four-digit random weight (or price) field. false instructs PAL not to calculate the check digit. Default value = false.
/RandWt5 /RW5	Boolean. true instructs PAL to automatically compute the internal check digit for a five-digit random weight (or price) field. false instructs PAL not to calculate the check digit. Default value = false.

Usage Notes

Valid characters for UPC-A bar codes are the digits 0 through 9. Any other characters will cause the string to be rejected with a rangecheck error.

UPC-A symbols require exactly 12 digits. If fewer digits are provided,, the string will be padded with leading zeros. If a check digit has been requested PAL computes the correct value and re-

places the 12th digit with the new check digit. The data string should contain a dummy character in the 12th digit to receive the check digit. Do not assume that only 11 digits are needed. Specifying short strings causes the string to be padded with zeros and the last character being replaced by the check digit.

If the RandWt4 or the RandWt5 flags are true, the last 4 or 5 digits before the check digit contain a random weight or a price. This allows in-store marking of such items as meat or cheese. A check digit for the random weight is calculated and replaces the character immediately preceding the 4- or 5-digit field. Since a new check digit must be calculated for the entire, symbol, the CheckDigit flag should always be set to true when using random weight symbols.

If the AddOn2 or the AddOn5 flags are true, a 2- or 5-digit add-on bar code is added to the right of the UPC-A symbol. Since the check digits for these add-on bar codes are implicit in the encoding, no check digit is specified for the add-on portion. To specify the text for a bar code with a supplement, simply add 2 or 5 digits to the end of a standard 12-digit UPC-A number. (Don't forget that the character before the supplement will contain the normal check digit.)

The generated image includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following example generates 1.13 inch tall UPC-A bar code with 0.013 inch narrow bars. A check digit will be calculated.

```
(012345678900) <</Height 72 1.13 mul /CheckDigit true>>
/UPCA _barcode
```

The following example generates 1.13 inch tall UPC-A bar code with 0.013 inch narrow bars and a 4-digit random weight field. Both zeros in the symbol will be replaced by check digits.

```
(212345012340) <</Height 72 1.13 mul /RandWt4 true
/CheckDigit true>> /UPCA _barcode
```

The following example generates 1.13 inch tall UPC-A bar code with 0.013 inch narrow bars and a 5-digit add-on bar code. Note that the zero in the symbol will be replaced with a check digit.

```
(1234567891012345) <</Height 72 1.13 mul /AddOn5 true
/CheckDigit true>> /UPCA _barcode
```

_barcode: UPC-E

DataStr [*CtrlDict*] **/UPCE _barcode**

This symbology conforms to the UPC Symbol Specification Manual.

Applicable Control Dictionary Entries

See appendix A., *Bar Code Considerations* for a discussion of the relationship between the CTM and default values.

/Height /H	Integer or fixed-point. Specifies the height, in current user units, of the overall bar code. For this bar code symbology, the height includes the human readable text. Default value = 36.0 (0.5" / 12.7 mm with default CTM).
/NarrowWidth /XWidth /NW /X	Integer or fixed-point. Specifies the width, in current user units, of the narrow bars. This is generally referred to as the X dimension. Default value = 0.936 (0.013" / 0.33 mm with default CTM).
/UPCE6 /E6	Boolean. true indicates that the data string contains a standard 12-digit UPC number. false indicates that zero-compressions has already been performed on the data string and the string contains exactly six digits. Default value = false.

Usage Notes

Valid characters for UPC-E bar codes are the digits 0 through 9. Any other characters will cause the string to be rejected with a rangecheck error.

If the UPCE6 flag is false UPC-E symbols require exactly the same 12-digit numbers as UPC-A. If fewer digits are provided,, the string will be padded with leading zeros. Since the check digit is integral in the bar encoding for UPC-E symbols, PAL always computes the correct value and replaces the 12th digit with the new check digit. The data string should contain a dummy character in the 12th digit to receive the check digit. Do not assume that only 11 digits are needed. Specifying short strings causes the string to be padded with zeros and the last character being replaced by the check digit.

12-Digit UPC numbers are zero-compressed using the UPC-E rules. If the number is not compressible (*e.g.*, does not contain enough central zeros), the string will be rejected with a rangecheck error.

If the 6-digit compressed value is known by the application, it may be specified directly if the UPCE6 flag is set to true. Since the check digit is computed internally, no dummy character should be left for it. Specify exactly 6 digits or less. If fewer digits than 6 are specified, the symbol is padded with leading zeros.

The generated image includes leading and trailing quiet zones (white space). The size of the quiet zone is 10 times NarrowWidth. If this results in a quiet zone of less than 0.10 inch (2.54 mm), the user should leave additional white space before and after the bar code.

The following examples generate 1.13 inch tall UPC-E bar codes with 0.013 inch narrow bars.

```
(12300000640) <</Height 72 1.13 mul>> /UPCE _barcode

(078349) <</Height 72 1.13 mul /UPCE6 true>> /UPCE _barcode
```

begin

Description

Pushes a dictionary onto the dictionary stack.

Usage

AnyDict **begin**

AnyDict Dictionary. Dictionary to push onto the dictionary stack.

Comments

The PAL interpreter searches the dictionary stack in order to locate values associated with names the interpreter encounters during PAL code execution. The interpreter always searches for names starting with the top-most dictionary on the dictionary stack. The interpreter continues searching down the stack until it locates the name. Therefore, higher dictionaries on the stack can contain definitions for names which supercede definitions in lower dictionaries.

The def operator allows the programmer to associate values with names within the top-most dictionary on the dictionary stack. During initialization, the PAL interpreter automatically places the special dictionary userdict on the top of the dictionary stack. This provides a default storage location for definitions performed during simple printer operations.

This def operator and name look-up construct provides the programmer with many powerful capabilities. However, the most common use of the construct involves the definition of simple variables.

The begin operator gives the programmer the ability to install a new dictionary on the top of the dictionary stack. The interpreter will then place any new definitions created by the def operator into this new top-most dictionary rather than into userdict.

This allows the programmer to collect a series of definitions within a single dictionary dedicated to that purpose. When the PAL code no longer requires the definitions, the programmer can use the end operator to discard the top-most dictionary from the stack. Provided the programmer has not created any secondary references to the dictionary, this has the affect of also discarding all definitions contained within the dictionary.

Hints

The begin and end operators provide the simplest means for procedures to manage local variables. The following example shows a very ineffecient procedure which averages two numbers. The procedure uses the begin and end operators to keep the variables First and Second local to itself.

```
1: /Average {
2:    <<>> begin
3:    /Second exch def
4:    /First exch def
5:    First Second add 2 div
6:    end
7: }  bind def
```

Line 2 places the procedure's empty dictionary onto the top of the dictionary stack. Lines 3 and 4 save the second and first parameters to the procedure under the variable names Second and First within the procedure's dictionary. Line 5 recalls the two values using the variable names and then performs the averaging equation. Line 6 discards the procedure's dictionary from the dictionary stack. Since no secondary reference to this dictionary exists, this also discards the First and Second variables from the printer's memory.

bind

Description

Optimizes the specified procedure for faster execution.

Usage

AnyProc **bind** *BoundProc*

AnyProc Procedure. The procedure to optimize.

BoundProc Procedure. The same procedure after optimization.

Comments

When initially sent to a PAL printer by a programmer, a procedure contains executable name objects which reference intrinsic operator objects in the dictionary stack. When the PAL interpreter executes the procedure, it must locate each name within the dictionary stack when it encounters the name in order to find the associated intrinsic operator object. The intrinsic operator object then instructs the interpreter to perform the desired action.

This process permits the programmer to substitute alternate definitions for the standard PAL interpreter operator names by associating alternate objects with each name on the dictionary stack. However, in most cases, the programmer does not require this capability. As a result, replacing the executable names within a procedure with the actual intrinsic operators associated with each name can improve the execution speed of a procedure. The bind operator performs this substitution.

Once the bind operator has optimized a procedure, the interpreter directly encounters the intrinsic operator objects when executing the procedure. The interpreter no longer needs to locate executable names within the dictionary stack every time it executes the procedure.

The bind operator also automatically optimizes any procedures defined within the specified procedure. The bind operator only substitutes executable name objects associated with intrinsic operator objects. The operator does not substitute name objects associated with numeric, string, or other object types.

Hints

Once the bind operator has replaced an executable name object with its associated intrinsic operator object within a procedure, changes to the executable name within the dictionary stack no longer affect the procedure. Therefore, the bind operator can protect a procedure from performing undesired actions by preventing subsequent changes to operator names from affecting the procedure.

bitshift

Description

Shifts the bits of an integer left for positive shift counts, and right for negative shift counts.

Usage

AnyInt ShiftInt **`bitshift`** *ShiftedInt*

AnyInt	Integer. Integer value consisting of bits to shift.
ShiftInt	Integer. Distance, in bits, over which to shift the bits of *AnyInt*. Positive values for *ShiftInt* result in left-shifts. Negative values result in right-shifts.

Comments

The interpreter will shift the bits of the specified integer, *AnyInt*, by the specified number of bits, *ShiftInt*. Positive values for *ShiftInt* instruct the interpreter to shift the bits to the left. Under the PAL specification, shifting left implies shifting toward the most significant bit of the value. Negative values for *ShiftInt* instruct PAL to shift the bits to the right — toward the least significant bit of the value.

The interpreter always fills any vacated bit positions with zero value bits. Filling with zeroes occurs regardless of the direction of the shift. Therefore, the interpreter always performs an unsigned shift operation.

ceiling

Description

Returns the next higher integer value.

Usage

AnyNum **`ceiling`** *CeilingNum*

AnyNum	Integer or fixed-point. Value to raise to the next higher integer.
CeilingNum	Integer or fixed-point. Next higher integer above *AnyNum*. The type of the returned value matches the type of the supplied parameter.

Comments

Although this operator will accept integer values, this operator has no affect upon integers. The following table shows the affect of the ceiling operator upon various fixed-point values.

```
 1.6  ceiling  2.0
 1.5  ceiling  2.0
 1.4  ceiling  2.0
 1.0  ceiling  1.0
 0.0  ceiling  0.0
-1.0  ceiling -1.0
-1.4  ceiling -1.0
-1.5  ceiling -1.0
-1.6  ceiling -1.0
```

clear

Description

Discards all objects from the operand stack.

Usage

NAny..1Any **`clear`**

NAny..1Any Any. All objects on the operand stack.

Comments

The clear operator discards all objects from the operand stack. The programmer will find this operator useful for ensuring a clean stack at the start of a job.

cleartomark

Description

Discards all objects from the operand stack down to, and including, the top-most mark object.

Usage

Mark NAny..1Any **`cleartomark`**

Mark Mark. Top-most mark object on the stack.

NAny..1Any Any. All objects on the operand stack above the top-most mark object.

Comments

The cleartomark operator discards all objects from the operand stack above the top-most mark object. The operator also discards the mark object itself.

By pushing a mark object onto the stack, the programmer can later restore the stack to that prior level simply by using the cleartomark operator.

closepath

Description

Closes the current drawing sub-path by drawing a line from the current point to the initial point of the sub-path.

Usage

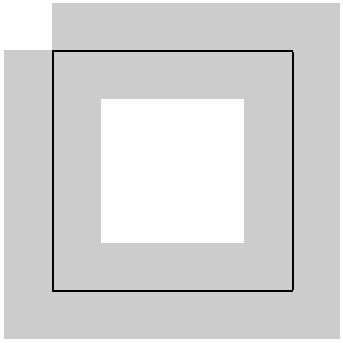
```
closepath
```

Comments

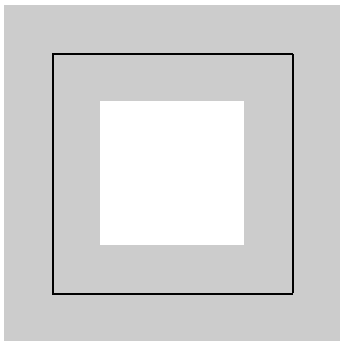
Paths consist of a series of line segments. As the programmer adds additional line segments onto the end of a path, PAL joins the lines in memory. By joining the lines, PAL can smooth the transition between two line segments during drawing. However, even if the programmer makes the end of the final line meet the start of the first line, PAL does not know if it should join together these first and last line segments.

By using the closepath operator, the programmer instructs PAL to connect the end of the path to the beginning of the path. This allows PAL to smooth the connection between the last and first line segments of the path.

The example below shows what happens when using only the moveto, lineto, and stroke operators to draw a square with butt line caps. The solid line in the center shows the path specified by the moveto and lineto operators. The gray area shows the line actually drawn based on the current line width. The drawing sequence starts and ends in the upper left corner of the square. Since the example does not use closepath, PAL draws the start and end of the path, both in the upper left corner, using the default butt line caps. This results in a notch where the lines come together. The size of the notch varies directly with the line width.



By replacing the final lineto operator with the closepath operator, the programmer instructs PAL to smooth the connection at the start and end of the square. The example given below illustrates the result of using the closepath operator instead of the final lineto operator necessary to close the square.



Hints

The programmer should always use the closepath operator instead of the final lineto operator to close the path. The closepath operator implies a lineto operation. Therefore, drawing a square involves a moveto operation, three lineto operations, and a final closepath operation. At a minimum, specifying four lineto operations followed by a closepath operation wastes time and a little memory. However, under certain circumstances, it may also produce undesired results.

concat

Description

Appends one string to the end of another string.

Usage

LeadStr TrailStr **concat** *ConcatStr*

LeadStr String. The string containing the characters which PAL will place at the start of the new *ConcatStr* string.

TrailStr String. The string containing the characters which PAL will place at the end of the new *ConcatStr* string.

ConcatStr String. The new string created by the interpreter which begins with the characters from *LeadStr* and ends with the characters from *TrailStr*.

Comments

The interpreter creates a the new string *ConcatStr*. *ConcatStr* will consist of the characters from *LeadStr* followed by the characters from *TrailStr*.

copy

Description

Duplicates multiple stack objects or copies one array, dictionary, or string to anther array, dictionary, or string.

Usage

NAny..1Any NInt	**copy**	*NAny..1Any NAny..1Any*
1Array 2Array	**copy**	*2Array*
1Dict 2Dict	**copy**	*2Dict*
1Str 2Str	**copy**	*2Str*

NAny..1Any Any. Stack objects to duplicate.

NInt Integer. Number of stack objects to duplicate.

1Array Array. Source array to copy to *2Array*.

2Array Array. Target array to receive copy of *1Array*.

1Dict Dictionary. Source dictionary to copy to *2Dict*.

2Dict Dictionary. Target dictionary to receive copy of *1Dict*.

1Str String. Source string to copy to *2Str*.

2Str String. Target string to receive copy of *1Str*.

Comments

The first variant duplicates the top *NInt* objects on the operand stack. For composite objects, the source and duplicate objects will share the same composite data.

The second variant duplicates the composite data within *1Array* to *2Array*. The objects within *1Array* replace the same position objects within *2Array*. *2Array* must match or exceed the size of *1Array*. If *2Array* exceeds the size of *1Array*, the operation will not affect the additional elements of *2Array*. If *1Array* contains composite objects, the duplicate objects placed into *2Array* will share their composite data with the original objects in *1Array*.

The third variant duplicates into *2Dict* the key + value entries within *1Dict*. The key + value pairs within *1Dict* will replace any key + value entries within *2Dict* which have identical keys. For unique key values, the interpreter will add the key + value entries from *1Dict* to *2Dict*. If *2Dict* contains keys which do not match any keys within *1Dict*, the operation will not affect those *2Dict* entry pairs. If *1Dict* contains composite objects, the duplicate objects placed into *2Dict* will share their composite data with the original objects in *1Dict*.

The fourth variant duplicates the characters in *1Str* into *2Str*. The characters within *1Str* replace the same position characters within *2Str*. *2Str* must match or exceed the size of *1Str*. If *2Str* exceeds the size of *1Str*, the operation will not affect the additional characters of *2Str*.

Hints

The following examples demonstrate the subtle difference between using the dup and copy operators. The first example shows the use of the dup operator to place a second reference to an array onto the stack. The second example shows the use of the copy operator to make a copy of an array into a new array. The second and third lines of each example shows the interpreter's output in response to the == operators.

```
1: [0 1 2 3] dup dup 1 (hello) put == ==
   [0 (hello) 2 3]
   [0 (hello) 2 3]

2: [0 1 2 3] dup dup length array copy dup 1 (hello) put == ==
   [0 (hello) 2 3]
   [0 1 2 3]
```

As shown by the first example, the dup operator simply creates a second reference to the same array. As a result, the put operator changes the single array referenced by both stack entries.

The second example uses the dup, length, and array operators to create a new array with the same number of elements as the original array. The example then uses the copy operator to copy all of the original array's contents to the new array. The put operator then only affects the new array.

Both the dup and copy operators serve seperate but equally useful functions. The dup operator creates a second reference to an array. Since both references share the same data, both references share any modifications made to the array. In addition, the programmer does not need to consume additional memory by have multiple copies of the same array.

The copy operator allows the programmer to create a new array containing the same data as the first array. This allows the programmer to modify either the first or second array without affecting the other array. However, each array consumes memory space within the printer. In addition, any composite objects copied from one array to the other will share their data.

count

Description

Returns a count of the number of objects currently on the operand stack.

Usage

NAny..1Any **count** *NAny..1Any NInt*

NAny..1Any Any. All objects on the operand stack.

NInt Integer. Number of objects on the operand stack..

Comments

The count operator pushes onto the operand stack a count of the number of objects resident on the operand stack prior to execution of the count operator.

counttomark

Description

Returns a count of the number of objects currently on the operand stack above the top-most mark object.

Usage

Mark NAny..1Any **count** *Mark NAny..1Any NInt*

Mark	Mark. Top-most mark object on the operand stack.
NAny..1Any	Any. All objects on the operand stack above the top-most mark object.
NInt	Integer. Number of objects on the operand stack above the top-most mark object.

Comments

The counttomark operator pushes onto the operand stack a count of the number of objects resident on the operand stack above the top-most mark object prior to execution of the counttomark operator. The count does not include the mark object itself.

currentdict

Description

Returns the dictionary object currently on the top of the dictionary stack.

Usage

`currentdict` *CurDict*

CurDict Dictionary. Dictionary object current on top of the dictionary stack.

Comments

currentdict pushes a dictionary object for the dictionary currently on top of the dictionary stack. The operator does not alter the dictionary stack. The dictionary remains on top of the dictionary stack. The operator pushes a duplicate object which references the same dictionary data.

Under most circumstances, this operator proves useful when deleting an entry made in the curent dictionary using the def operator. By using "currentdict */Name* undef", the user does not need to know which dictionary resides on top of the dictionary stack. The user only needs to know that no other dictionary was pushed (using begin) onto the dictionary stack since def was used to define */Name*.

currentgray

Description

Returns the current color setting mapped to the DeviceGray color space.

Usage

`currentgray` *LevelFxpt*

LevelFxp Fixed-Point. Current color setting mapped to the DeviceGray color space. The value ranges from 0.0 (black) to 1.0 (white).

Comments

PAL currently only supports the DeviceGray color space. This operator will return the last gray level established using the setgray operator.

Different printer models may have different interpretations for the gray level requested via the setgray operator. PAL printer models which support only black and white printing will always translate gray level requests into either black or white.

The currentgray operator always returns the color *requested* via the setgray operator and not the color which the printer may actually be printing. Therefore, when the user requests 20% gray level by specifying "0.2 setgray" on a printer which supports only black and white, the printer will probably round the 20% gray level request down to 0% gray level (black). However, the currentgray operator will still return 0.2 (20%) since that was the gray level requested via the setgray operator.

PAL printers default at power-on to 0% gray level (black).

currentpoint

Description

Returns the coordinates of the current point.

Usage

`currentpoint` *XNum YNum*

XNum Integer or fixed-point. Current X coordinate in user coordinates.

YNum Integer or fixed-point. Current Y coordinate in user coordinates.

Comments

The operator returns the position of the current point. PAL returns the coordinates in the user coordinate system.

Some PAL drawing applications move the current point. For example, the show operator automatically moves the current point to the end of drawn string. This allows the user to quickly draw another string following the first string without having to calculate the string's proper position. If required, the currentpoint operator allows the programmer to determine the new current point location following these draw operations.

CVS

Description

Converts a numeric value to a string containing the value's human readable decimal representation.

Usage

ValNum DummyStr **cvs** *DecStr*

ValNum Integer or fixed-point. Numeric value to convert into a string.

DummyStr String. Required for historical compatibility. The interpreter does not use this parameter.

DecStr String. String created by the interpreter which contains the human readable decimal representation of *ValNum.*

Comments

The interpreter converts *ValNum* to its human readable decimal representation. If then creates *DecStr* containing the result.

For positive values, *DecStr* will not contain a leading plus (+) sign. *DecStr* will contain a leading negative (-) sign for negative *DecStr* values. *DecStr* will not include any leading zeroes or commas. For fixed-point values, *DecStr* will contain a decimal point (.) with at least one digit both before and after the decimal point.

The following examples demonstrate some of the results possible using the cvs operator.

```
   0.1 ()   cvs   (0.1)
  -1.0 ()   cvs   (-1.0)
+00.01 ()   cvs   (0.01)
-10.00 ()   cvs   (-10.0)
    45 ()   cvs   (45)
   -14 ()   cvs   (-14)
```

CVX

Description

Converts a literal name or file object into a executable object.

Usage

LiteralFile **cvx** *ExecFile*
LiteralName **cvx** *ExecName*

LiteralFile File. Literal file object to convert to executable.

ExecFile File. Literal file object converted to executable.

LiteralName Name. Literal name object to convert to executable.

ExecName Name. Literal name object converted to executable.

Comments

When PAL encounters an object for execution, it checks the object's literal/executable attribute. If the object has a literal attribute, PAL simply pushes the object onto the top of the operand stack. If the object has an executable attribute, PAL attempts to perform any operations stored within the object.

Under most circumstances, PAL automatically assigns objects the literal attribute. This means that PAL will treat the objects as data whenever it encounters the objects.

Name Objects

Executable names provide the most notable exception to this rule. PAL requires the placement of a slash character at the start of a name in order to specify a literal name. Without the preceding slash character, PAL will treat a name as executable.

Programmers often find it beneficial to store within a database various references to procedures. For example, the records within a parts database could include references to a procedure which draws the picture of the part.

In order to store the name of each procedure within the database, the programmer must specify a literal name. This prevents PAL from attempting to immediately execute the procedure. The cvx operator allows the programmer to convert the literal name object to executable in order to execute the associated procedure.

Once the programmer has converted a literal name to executable using the cvx operator, the programmer must then use the exec operator to instruct the interpreter to execute the converted name.

File Objects

The file and _devicefile operators return literal file objects. These objects provide a type of pointer to the opened file. When the programmer wishes to read or write the file, the programmer

must specify the file object associated with the file. By specifying the file object, the programmer informs PAL which file to read or write.

Normally, the PAL interpreter reads the standard file %stdin to receive operations to perform. However, the programmer can use the exec operator to specify an different file for the interpreter to read. However, the exec operator requires that the programmer specify an executable file object and not a literal file object. Therefore, the programmer must use the cvx operator to convert the literal file objects returned by file and _devicefile to executable.

Hints

The cvx operator provides the programmer with very advanced capabilities which only the most advanced applications will require.

As mentioned above, the programmer can store the names of procedures within a database constructed from array and/or dictionary objects. The following provides an example of this type of database.

```
1: /WidgetProc {36 36 moveto (Widget) show} bind def
2: /BobbelProc {36 36 moveto (Bobbel) show} bind def
3: /Parts <<
4:   /Widget [12.95 6 /WidgetProc]
5:   /Bobbel [99.95 8 /BobbelProc]
6: >> def
7:
8: Parts /Widget get 2 get cvx exec showpage
9: Parts /Bobbel get 2 get cvx exec showpage
```

Lines 1 and 2 define unique page drawing procedures for each part in the Parts database. Lines 3 through 6 define the actual parts database. Each database entry contains the name of the part as the key, and an array of data associated with each part. The array contains the part's price, the quantity on hand, and the name of the part's page drawing procedure. Lines 8 and 9 recall each part from the Parts database and print the page associated with the part.

def

Description

Stores a data value under a specified key in the top-most dictionary on the dictionary stack.

Usage

KeyAny DataAny **`def`**

KeyAny Any type. Key under which to store the specified object.

DataAny Any type. Object to store under the specified key.

Comments

Both *KeyAny* and *DataAny* may be of any type. However, the interpreter provides special optimized handling of name objects when used as dictionary entry keys. As a result, if the programmer specifies a string for *KeyAny*, the interpreter automatically converts the string to a literal name before storing the key into the dictionary. In addition, names provide the easiest means for recalling the *DataAny* object at a future time.

The programmer can use the def operator to save any data object, including complex data objects such as dictionaries, arrays, and procedures into the printer's memory. If a name object was used for *KeyAny*, the programmer or procedures written by the programmer can later recall these data objects onto the operand stack by simply specifying *KeyAny*.

If an entry for *KeyAny* already exists in the top dictionary on the dictionary stack, def replaces the old *DataAny* in the dictionary's entry with the new *DataAny*.

During printer initialization, the PAL interpreter creates an empty dictionary named userdict. The interpreter then places this dictionary on the top of the dictionary stack. Unless the PAL programmer places another dictionary above userdict on the dictionary stack, the def operator will store *KeyAny* and *DataAny* into userdict. The PAL interpreter provides this dictionary for exactly this purpose. userdict provides an easily accessible location for storing data objects and procedures.

Hints

Using name objects as the *KeyAny* parameter serves the same basic purpose as variables in other programming languages. The def operator provides the same basic purpose as assigning a value to a variable. However, the def operator also provides the means to store PAL procedures within the printer's memory. The host computer can later invoke these procedures to perform various desired actions. Other procedures can also invoke these procedures in a manner similar to using subroutines or functions in other programming languages.

_devicefile

Description

Opens a device for reading and/or writing at a low access level.

Usage

FileStr AccessStr **_devicefile** *OpenFile*

FileStr String. Specifies the name of the device to open.

AccessStr String. Specifies the type of access to the device which the programmer desires. See the file operator discussion for information on *AccessStr*.

Comments

Only a very small percentage of PAL applications require the use of the _devicefile and file operators. Out of that small percentage, an even smaller percentage require the use of the _devicefile operator.

When used with storage devices such as flash memory, the _devicefile operator normally by-passes the printer's standard file management services. The operator allows the programmer to directly access the storage medium. Especially in the case of performing any write operations, this can result in irreparable damage to file management information contained on the storage media. Thereby rendering any files contained on the media inaccessible.

Only experienced programmers with detailed information regarding the printer's use of storage media should attempt to use the _devicefile operator.

_deviceformat

Description

Clears a low level device such as flash memory.

Usage

FileStr AccessStr **`_deviceformat`**

FileStr String. Specifies the name of the device to open.

AccessStr String. The access string for this operator must be an empty string, i.e. ().

Comments

This operator causes the storage device being accessed to be completely cleared. CAUTION must be used with this operator since any pre-loaded PAL applications in the specified device file will be lost.

Only experienced programmers with detailed information regarding the printer's use of storage media should attempt to use the _deviceformat operator.

dict

Description

Creates an empty dictionary.

Usage

PairsInt **dict** *EmptyArray*

PairsInt Integer. Number of key + value entry pairs anticipated for the dictionary.

EmptyDict Dictionary. An empty dictionary.

Comments

This operator performs the exact same function as the PAL sequence "<<>>". The operator requires the *PairsInt* parameter strictly for historical compatibility. Historically, the *PairsInt* parameter specified the number of key + value pairs which the programmer anticipated the dictionary to hold. Although the parameter no longer affects the operator, the *PairsInt* parameter is required from a syntactical standpoint.

div

Description

Divides the next-to-top stack value by the top stack value and returns the quotient.

Usage

DividendNum DivisorNum **`div`** *QuotientFxp*

DividendNum Integer or fixed-point. Value to divide by *DivisorNum*.

DivisorNum Integer or fixed-point. Value to divide into *DividendNum*.

QuotientFxp Fixed-point. Result of division. The interpreter always returns a fixed-point result regardless of the operands.

Comments

The div operator always generates a fixed-point quotient regardless of the operand types. PAL provides the idiv operator for calculating integer quotients.

Hints

As with most computer systems, most PAL printers can perform integer calculations faster than fixed-point calculations. Therefore, the programmer should consider using integer math whenever possible. The idiv operator performs an integer divide rather than a fixed-point divide. As a result, the programmer should consider using idiv instead of div whenever possible.

_dspclear

Description

Clear an area of the printer's front panel character display.

Usage

BBoxArray **_dspclear**

BBoxArray Array. Bounding box for area to clear.

[*Left Top Right Bottom***]**

Left Integer or fixed-point. Left-most character column of display area to clear. The interpreter includes this column as part of the area cleared.

Top Integer or fixed-point. Top-most character line of display area to clear. The interpreter includes this line as part of the area cleared.

Right Integer or fixed-point. Right-most character column of display area to clear. The interpreter includes this column as part of the area cleared.

Bottom Integer or fixed-point. Bottom-most character column of display area to clear. The interpreter includes this line as part of the area cleared.

Comments

The *BBoxArray* parameter specifies the bounding box of the display area which the operator will clear. The *BBoxArray* array object must contain the four values shown above.

These four values establish a rectangular region of character positions on the display for the operator to clear. *Left* specifies the left-most character column to include as part of the clear operation. *Right* specifies the right-most character column to include. *Top* specifies the top-most line to include. *Bottom* specifies the bottom-most line to include.

Although variations may occur between different PAL printers, in general column 0 specifies the left-most column of the display and row 0 specifies the top-most row. The column numbers increment to the left, and the row numbers increment down.

_dspmovecursor

Description

Reposition visible cursor on front panel character display.

Usage

ColumnNum LineNum **_dspmovecursor**

ColumnNum Integer or fixed-point. Character column at which to locate visible cursor.

LineNum Integer or fixed-point. Character line at which to locate visible cursor.

Comments

Most PAL printers with front panel displays have the ability to display a cursor on the display. Presentation of a cursor can prove very useful when requesting input from the printer operator via the front panel. The _dspmovecursor operator provides the PAL programmer with the ability to locate this cursor where ever appropriate.

The PAL interpreter maintains separate locations for the visible cursor and the invisible next character position. This allows the PAL programmer to write new characters onto the display without affecting the position of the visible cursor.

Under normal printer operation, the PAL interpreter leaves the displayable cursor disabled. This has the affect of removing the cursor from the display. The PAL programmer can freely move the display cursor around the display. However, the operator will not see the cursor until the programmer enables the cursor via the _dspsetcursor operator. Once the programmer enables the cursor, it will appear at the last cursor position established via the _dspmovecursor operator.

_dspmoveto

Description

Reposition invisible next character pointer on front panel character display.

Usage

ColumnNum LineNum **_dspmoveto**

ColumnNum Character column at which to locate invisible next character pointer.

LineNum Character line at which to locate invisible next character position.

Comments

On printers with front panel character displays, _dspmoveto allows the programmer to position the invisible next character pointer on the display. The invisible next character pointer establishes the location at which the _dspstring operator will display any future string.

The PAL interpreter maintains separate locations for the invisible next character pointer and the visible cursor. This allows the PAL programmer to write new characters onto the display without affecting the position of the visible cursor.

_dspsetcursor

Description

Select front panel character display visible cursor style.

Usage

ControlDict **_dspsetcursor**

ControlDict Dictionary. Controls for establishing new visible cursor style.

/Block Boolean. true enables the displaying of a block style cursor. false disables the block style cursor. Enabling the block style cursor does not necessarily disable any other cursor style. Some displays have the ability to display multiple cursor styles at one time.

/DspEnable Boolean. true enables the front panel character display. false disables (blanks) the display.

/UL Boolean. true enables the displaying of an underline style cursor. false disables the underline style cursor. Enabling the underline style cursor does not necessarily disable any other cursor style. Some displays have the ability to display multiple cursor styles at one time.

Comments

Some displays only support a single cursor style. If that is the case, then the /Block and /UL functions behave the same.

_dspstring

Description

Displays text at the invisible next character position on the front panel character display.

Usage

AnyStr **_dspstring**

AnyStr String. Text to display.

Comments

This operator provides the primary means for programmers to display messages on the printer's front panel character display.

Programmers should expect to find this operator only on printers with front panel displays. The discussion of the _dspclear operator includes important information concerning PAL's support for front panel displays.

dup

Description

Pushes a second copy of the top-most object on the operand stack.

Usage

Any **dup** *Any Any*

Any Any. Stack object to duplicate.

Comments

This operator performs the same function as the PAL sequence "1 copy". It simply duplicates the top-most object on the operand stack. For composite objects, the source and duplicate objects will share the same composite data.

Hints

The copy operator discussion includes examples which show the subtle difference between copy and dup when duplicating composite objects.

end

Description

Pops the top-most dictionary from the dictionary stack.

Usage

end

Comments

The end operator removes dictionaries from the dictionary stack placed there by the begin operator. The begin operator discussion also covers the end operator.

eq

Description

Compare two objects for equality.

Usage

1Any 2Any **eq** *Bool*

1Any	Any. First object to compare. With the exception of integer, fixed-point, string, and name objects, *1Any* must have the same object type as *2Any*. The operator will compare integer and fixed-point objects in any combination. The operator will also compare any combination of string and name objects.
2Any	Any. Second object to compare.
Bool	Boolean. A value of true indicates equality. A value of false indicates inequality.

Comments

PAL will compare for equality any two objects of the same object type. PAL will also compare any combination of integer and fixed-point objects, as well as any combination of string and name objects.

PAL compares strings and names using the standard ASCII character sorting sequence including case sensitivity. Therefore, the string (abc) does not match the string (ABC).

For composite objects, the two objects must reference the exact same composite data. Therefore, the PAL sequence "[1 2 3] [1 2 3] eq" produces the result "false". However, the sequence "[1 2 3] dup eq" produces the result "true".

In the first case, the sequence creates two unique arrays which happen to contain the same data. Since the array objects reference different data within the printer's memory, the objects do not meet PAL's condition for equality.

In the second case, the sequence creates a single array and a second reference to the same array data. Since the array objects reference the same data within the printer's memory, the objects meet PAL's condition for equality.

Hints

PAL uses the same conditions for equality for the eq operator as it does when comparing key values in dictionaries.

erasepage

Description

Discard all drawing previously performed on the current page.

Usage

```
erasepage
```

Comments

Under most circumstances, programmers will have no use for this operator. The operator has the affect of canceling all previous drawing performed on the current page.

exch

Description

Exchange the top-most object on the operand stack with the next lower object.

Usage

1Any 2Any **exch** *2Any 1Any*

1Any Any. Second-to-top object on the operand stack.

2Any Any. Top-most object on the operand stack.

Comments

exch simply exchanges the positions of the two top objects on the operand stack.

exec

Description

Execute the object on the top of the operand stack.

Usage

Any **exec**

Any Any. Object to execute.

Comments

As PAL encounters objects received from the host computer or contained in procedures, PAL executes each object. In most cases, when PAL encounters an object, it marks the object as literal. This means that PAL will treat the object as data. When PAL executes a literal object, it does nothing more than push the object onto the top of the operand stack. Therefore, PAL simply pushes most of the objects it encounters onto the operand stack.

The exec operator instructs PAL to "encounter" the object on the top of the object stack. As a result, PAL pops the object off the stack and pretends it has just received the object from the host computer. This causes the interpreter to execute the object. Just as in the case of objects received from the host, literal objects on the top of the stack will simply result in PAL pushing the object back onto the top of the stack.

However, it PAL encounters an executable object on the stop of the stack. It will perform the operations associated with that executable object. Executable objects typically include name and file objects which the programmer has converted from literal to executable using the cvx operator. The cvx operator discussion includes using the exec operator in conjunction with the cvx operator.

_execexit

Description

Terminates executive mode and returns the printer to normal host communications mode.

Usage

```
_execexit
```

Comments

Although well suited for experimenting with PAL operators, PAL's executive mode generates prompts and other extraneous output which generally prove undesirable when communicating directly with a host computer. If the programmer uses the executive keyword to enter executive mode, the programmer can later use the _execexit keyword to terminate executive mode and reestablish normal host communications.

execform

Description

Captures the results of a drawing sequence for faster reuse on the same or subsequent pages.

Usage

FormDict **execform**

FormDict Dictionary. Contains parameters for capturing the desired drawing sequence as well as the procedure which performs the drawing sequence. This dictionary contains the following entries.

/BBox [*LeftNum BottomNum RightNum TopNum*]
Array. Specifies the area on the page in which PAL will capture any drawing operations. PAL will clip any drawing operations which exceed these boundaries. The values are in the form coordinate system, which is establish by applying the /Matrix entry in *FormDict* to the current coordinate system.

LeftNum Integer or fixed-point. Specifies the left edge of the capture area.

BottomNum Integer or fixed-point. Specifies the bottom edge of the capture area.

RightNum Integer or fixed-point. Specifies the right edge of the capture area.

TopNum Integer or fixed-point. Specifies the top edge of the capture area.

/FormType Integer. This entry exists primarily to provide for future expansion. This entry must have the value one (1) at this time.

/Matrix Array. Specifies a transformation matrix which PAL will apply (concatenate to) the current transformation matrix before starting the drawing operations to be captured. To use the current transformation matrix setting, specify a matrix of "[1 0 0 1 0 0]".

/PaintProc Procedure. Specifies the procedure containing the drawing operations which PAL will capture.

Comments

The execform operator instructs PAL to capture a series of drawing operations. When PAL captures the specified drawing sequence, PAL produces a "form". A form is equivalent to a graphic image sent from a host computer. However, in the case of a form, the image is created within the printer's memory. PAL automatically determines the size of the image from the /BBox entry specified in *FormDict*.

After PAL captures the drawing sequences and has created the form, PAL saves the form image inside *FormDict*. Whenever PAL executes the execform operator, it checks the specified *FormDict* to see if the dictionary already contains a previously created form. If the dictionary already contains a form, PAL does not bother to execute the drawing sequences. Instead, PAL simply renders the already created form onto the page.

The execform operator provides the user with the capability of drawing repetative images only a single time. These images can then be rendered multiple times onto a single or multiple pages without having to repeat the drawing operations. This has the affect of improving print speed since the printer does not have to perform redundant drawing operations.

Although the execform operator has several complex parameters, the operator can be easy to use if a couple simple models are followed. First, in all cases, the /FormType entry always has the value one (1). Therefore, every *FormDict* must contain the entry "/FormType 1". Other values have been reserved for future options.

Second, most users do not wish to alter the current transformation matrix when drawing their form. Most users wish to draw their form using the same coordinate system in which they are drawing the rest of their page. As a result, *FormDict* typically contains the entry "/Matrix [1 0 0 1 0 0]". This entry specifies that the current transformation matrix will not be altered for the drawing operations.

The /BBox entry in *FormDict* specifies the area on the page in which the form image is to be drawn. The first time execform is executed, PAL will capture all drawing operations which occur to that area of the page. It will then save the image of that area as the form image within *FormDict*. If *FormDict* already contains a previous form image when execform is executed, PAL will then draw the existing image onto the page within the specified /BBox area.

The /PaintProc entry in *FormDict* specifies the actual procedure which performs the drawing operations to be captured within the form image. The entry has an actual procedure, and not just the name of a defined procedure. Therefore, the entry has the appearance "/PaintProc {*...drawing operations...*}". However, for more complex drawing operations, it is common for the /PaintProc procedure simply to execute a separately defined procedure. For example, the entry might simply be "/PaintProc {MyFormDrawProc}", where MyFormDrawProc is the name of a previously defined procedure.

Since PAL saves the form image within *FormDict*, the user should treat *FormDict* as if it were the actual form image. So long as the user keeps *FormDict* saved in memory, any form image contained in *FormDict* will also remain saved. Once the user discards *FormDict* from memory, the form image will be lost and will need to be regenerated if it is required in the future.

Since form images can consume considerable memory depending upon their size, users should consider explicity discarding any *FormDict*s from memory when changing between different types of pages to be printed. This will release the memory being used for form images which do not apply to the new pages.

The following shows a template for one way to use the execform operator. In the template, the user simply needs to replace the *italicized* portions with the information appropriate for the form to be rendered.

```
/FormNameProc {
  ...form drawing instructions...
} bind def

/FormNameDict <<
  /FormType 1
  /BBox [Left Bottom Right Top]
  /Matrix [1 0 0 1 0 0]
  /PaintProc {FormNameProc}
>> def

FormNameDict execform
```

First, the user should select some name for the form. This name should replace *FormName* in the above template. This will result in the definition of a procedure under the selected form name with **Proc** appended to the end of the form name. It will also result in the definition of a dictionary to contain the form image under the user's selected form name with **Dict** appended to the end of the form name.

The definition of the *FormName***Proc** contains the operations necessary to render the form image onto the page. These operations are indicated by "*...form drawing instructions...*" in the template.

Left, Bottom, Right, and *Top* in the template specify the area on the page in which the form is to be drawn. These values are specified using the current coordinate system. Since PAL defaults to typesetter points for the coordinate system, these values should be specified in typesetter points unless the user has altered the coordinate system.

Within the form dictionary *FormName***Dict**, the **/PaintProc** entry has been simplified to execute the separate procedure *FormName***Proc**. This allows the user to define a large series of drawing operations outside the definition of the form dictionary.

Whenever the user wishes to render the form onto the page, the user simply needs to specify the operation "*FormName***Dict execform**". The first time PAL encounters the operation *FormName***Dict** will not yet contain a form image, so PAL will execute **/PaintProc** which in turn will execute */FormName***Proc**. PAL will then capture the form image and save it as part of *FormName***Dict**. For all subsequent executions of "*FormName***Dict execform**" PAL will simply use the form image already contained within *FormName***Dict**.

The user must be careful not to redefine *FormName***Dict** once it has been defined. If the user redefines *FormName***Dict**, any form image contained within *FormName***Dict** will be lost and PAL will have to recreate it during the next **execform** operation involving that dictionary. Therefore, the definition of *FormName***Dict** as well as *FormName***Proc** should only occur as part of a print job initialization sequence before the sequence of pages starts printing.

At the end of the print job, the user can use the undef operator to discard *FormName***Dict** and any form image it contains. This can release a significant amount of memory when larger form images are involved.

executive

Descriptions

Places the printer into a prompted line input mode to facilitate experimentation with PAL operators using a terminal or terminal emulation program.

Usage

```
executive
```

Comments

executive mode makes it easier for a programmer to directly interface with the PAL interpreter using a terminal or terminal emulation program. Under executive mode, the interpreter will prompt the programmer when the interpreter requires additional input.

Once prompted, the programmer can enter a full line of PAL data and operators. During line input, the programmer can use backspace and other limited line editing capabilities. Under executive mode, the interpreter accepts the following line editing controls.

Oct	Dec	Hex	Mnemonic	Typical Keys	Description
010	008	08	BS	Ctrl+H	*Backspace.* Delete last character on the line.
012	010	0A	LF	Ctrl+J	*Line Feed.* Indicates completion of the current line, thereby allowing PAL to process the line.
015	013	0D	CR	"Enter", "Return", or Ctrl+M	*Carriage Return.* Functions same as Ctrl+J.
022	018	12	DC2	Ctrl+R	*Retype.* PAL will display the line as it has been entered so far.
025	021	15	NAK	Ctrl+U	*Undo.* Instructs PAL to discard the current line so that the programmer can start the line over again.

Under executive mode, the interpreter continues to process data in exactly the same manner as under normal host communications. The only difference being that the interpreter allows the programmer to enter and edit a full line before processing the data.

Entering and exiting executive mode has no effect upon other areas of printer operation. The programmer can start by sending commands to the printer under normal host communications mode. The programmer can then switch to executive mode to supply additional commands. The programmer can then use the _execexit operator to terminate executive mode and resume normal host communications.

exit

Descriptions

Terminate the inner-most active loop.

Usage

```
exit
```

Comments

The exit operator allows the programmer to prematurely terminate a loop. The operator terminates only the inner-most active loop. Since the loop operator includes no termination condition for the loop it creates, the programmer must use the exit operator to terminate a loop created by the loop operator. The exit operator will also terminate loops created by the for and repeat operators.

file

Description

Opens a data file for reading and/or writing.

Usage

FileStr AccessStr **`file`** *OpenFile*

FileStr String. Specifies the name of the file to open.

AccessStr String. Specifies the type of access to the file which the programmer desires. PAL currently supports the value *AccessStr* values.

(a) Write-only access. If the file exists, the file's write pointer will be automatically set to the end of the file (append). If the file does not exist, the file will be created.

(a+) Read/write access. If the file exists, the file's read/write pointer will be automatically set to the end of the file (append). If the file does not exist, the file will be created. Same as (a) except read access permitted.

(r) Read-only access. If the file exists, the file's read pointer will be automatically placed at the start of the file. An error will result if the file does not already exist.

(r+) Read/write access. If the file exists, the file's read/write pointer will be automatically placed at the start of the file. An error will result if the file does not already exist. Same as (r) except write access permitted.

(w) Write-only access. If the file exists, it will be truncated to zero bytes in length. If the file does not already exist, the file will be created.

(w+) Read/write access. If the file exists, it will be truncated to zero bytes in length. If the file does not already exist, the file will be created. Same as (w) except read access permitted.

OpenFile File. File object associated with file just openned.

Comments

This operator provides the programmer with the ability to access data files on printers which support data file storage. Data file storage can vary greatly from one PAL printer to the next. Some printers may include floppy or hard disk drives. Other printers may include solid-state memory cards. The programmer should consult each printer's documentation for information concerning available data file storage.

After opening the requested file, PAL returns the file object *OpenFile* on the top of the stack. Since the programmer can have an indefinite number of files open simultaneously, the PAL operators which access files require the programmer to supply a file object as a parameter. The file object tells each operator which file to access.

All PAL printers recognize the following standard file names.

%stdin — Standard input file. Normally the communications port connected to the host computer. Read-only access.

%stdout — Standard output file. Normally the communications port connected to the host computer. Write-only access.

%stderr — Standard error output file. Normally the same as %stdout. Write-only access.

Some PAL printers, depending on their available features, will recognize various combinations of the following standard files.

%keybrd — Keyboard. Normally associated with a PS/2 style keyboard. Usually an add-on option available for the printer. Read-only access.

%keypad — Keypad. Normally associated with a printer's front panel key pad.

A file remains open for as long as the programmer maintains any reference to the file's associated file object. PAL automatically closes a file once the programmer eliminates all references to the file's associated file object. The following examples demonstrate the file open and close process.

```
1:  (%stdin) (r) file
2:  (123) readstring == ==
```

```
1:  <<>> begin
2:    /HostRead (%stdin) (r) file def
3:    HostRead (123) readstring == ==
4:    HostRead (123) readstring == ==
5:  end
```

The first example attempts to read three characters from the %stdin file. Line 1 opens the file. At the end of line 1, PAL has left the file's associated file object on the top of the stack. Line 2 uses this file object, plus the string required by the readstring operator to perform a read from %stdin. The == operators write the results of the readstring operation to %stdout. readstring removes the file object and the required string from the stack. Since this eliminates all references to the file object from the printer's memory, PAL automatically closes the file upon completion of the readstring operation.

The second example attempts to read six characters from the %stdin file. Line 1 places an empty dictionary on the top of the stack to host the HostRead variable. Line 2 opens the %stdin file and saves the associated file object under the name HostRead in the previously empty dictionary. Lines 3 and 4 then recover this file object and attempt to read three characters from %stdin. Line 5 discards the temporary dictionary from the dictionary stack. Since that dictionary contained the only reference to the %stdin file object, PAL automatically closes the file after discarding the dictionary.

fileposition

Description

Pushes the offset of a file's read/write pointer onto the operand stack.

Usage

OpenFile **fileposition** *PositionInt*

OpenFile File. File object for open file from which to return file's read/write offset.

PositionInt Integer. Offset of file's read/write pointer from start of file. A value of zero indicates the pointer points to the first byte of the file.

Comments

This operator provides the programmer with the means of retrieving and, if desired, remembering a given file position. The programmer can then perform other file operations which may relocate the file read/write pointer. Upon completion of these operations, the programmer can restore the pointer to the remembered position by using the setfileposition operator.

Hints

The following example will remember the current position in MyFile, write the string "Hello" at offset 23 in the file, and then restore the pointer to the original file position.

```
MyFile fileposition
MyFile 23 setfileposition
MyFile (Hello) writestring
MyFile exch setfileposition
```

findfont

Description

Locates a font in the font resource directory.

Usage

FontName **`findfont`** *FontDict*

FontName Literal name. Name assigned to font in the font directory.

FontDict Dictionary. Copy of font's dictionary from the font directory. PAL creates a new dictionary containing the same information as the font's dictionary in the font directory. However, the programmer and PAL can add or delete entries in this new dictionary without affecting the original font dictionary.

Comments

PAL creates a copy of the font dictionary contained in the font directory so that the programmer may apply scaling and other alterations to the font. Although PAL makes a copy of the original font dictionary, the new dictionary shares its objects with the original dictionary. This means that the programmer may add and delete entries within the new dictionary. However, the programmer should not attempt to alter any of the objects already present within the dictionary. If existing objects require modification, the programmer must completely replace the original object with a new object.

floor

Description

Returns the next lower integer value.

Usage

AnyNum **`floor`** *FloorNum*

AnyNum	Integer or fixed-point. Value to reduce to the next lower integer.
FloorNum	Integer or fixed-point. Next integer value at or below *AnyNum*. The type of the returned value matches the type of the supplied parameter.

Comments

Although this operator will accept integer values, this operator has no affect upon integers. The following table shows the affect of the floor operator upon various fixed-point values.

```
 1.6  floor  1.0
 1.5  floor  1.0
 1.4  floor  1.0
 1.0  floor  1.0
 0.0  floor  0.0
-1.0  floor -1.0
-1.4  floor -2.0
-1.5  floor -2.0
-1.6  floor -2.0
```

for

Description

Performs a procedure for a specified number of iterations.

Usage

StartNum IncNum StopNum AnyProc **for**

StartNum Integer or fixed-point. Specifies the starting value for a counter maintained by the interpreter for the loop.

IncNum Integer or fixed-point. Specifies the incrementing value for a counter maintained by the interpreter for the loop.

StopNum Integer or fixed-point. Specifies the stopping value for a counter maintained by the interpreter for the loop.

AnyProc Procedure. The procedure executed by the interpreter following every increment of the counter.

Comments

The interpreter maintains an internal counter associated with each for loop encountered. The for loop starts by setting this internal counter to the specified *StartNum* value. Then, if the counter does not exceed *StopNum*, the interpreter pushes the counter value onto the stack and executes *AnyProc*. Upon completion of *AnyProc*, the interpreter adds *IncNum* to the counter. This process continues until the counter exceeds *StopNum*.

The interpreter always checks to see if the counter exceeds *StopNum* before executing the procedure. This means that the interpreter will never execute the procedure if *StartNum* exceeds *StopNum*.

PAL allows both positive and negative values for *IncNum*. If the programmer specifies a positive value for *IncNum*, the loop will terminate when the counter exceeds *StopNum* in a positive direction. In other words, when the counter is greater than *StopNum*.

If the programmer specifies a negative value for *IncNum*, the loop will terminate when the counter exceeds *StopNum* in a negative direction. In other words, when the counter is less than *StopNum*.

Before every execution of the procedure, PAL pushes the current counter value onto the stack. The programmer has the responsibility for removing these values from the stack. In many cases, the procedure will consume the value as part of its normal processing. However, if the procedure does not require the counter value, the procedure should include a pop operator to remove the value from the stack.

Hints

The programmer can also use the for operator to accumulate a series of values on the stack for inserting into an array or other data object. For example, the following PAL code uses an empty procedure to create an array containing all the even numbers from 14 to 114, inclusive.

```
[14 2 114 {} for]
```

ge

Description

Determine whether the first object is greater than or equal to the second object.

Usage

Any1Num Any2Num **ge** *Bool*
Any1Text Any2Text **ge** *Bool*

Any1Num Integer or fixed-point. First numeric object to compare.

Any2Num Integer or fixed-point. Second numeric object to compare.

Any1Text String or name. First text object to compare.

Any2Text String or name. Second text object to compare.

Bool Boolean. A value of true indicates the first object meets or exceeds the second object. A value of false indicates the second object exceeds the first object.

Comments

In the first variant, the operator determines whether or not the first numeric parameter is greater than or equal to the second numeric parameter. The operator will accept integer and fixed-point objects for either parameter.

In the second variant, the operator determines whether or not the first text parameter is greater than or equal to the second text parameter. The operator will accept string or name objects for either parameter. PAL compares strings and names using the standard ASCII character sorting sequence including case sensitivity. Therefore, the string (Abc) is less than the string (abc).

get

Description

Recover data from a composite or string object.

Usage

```
AnyArray IndexInt   get   ElementAny
 AnyDict KeyAny     get   ValueAny
 AnyStr IndexInt    get   CharInt
```

AnyArray Array. Array object containing desired object to recover.

AnyDict Dictionary. Dictionary object containing desired object to recover.

AnyStr String. String object containing desired character to recover.

IndexInt Integer. When used with *AnyArray*, index of data object within array. Arrays begin with index zero. When used with *AnyStr*, index of character within string. Strings begin with index zero.

KeyAny Any. Key associated with value object within dictionary.

ElementAny Any. Object from index *IndexInt* within array *AnyArray*.

ValueAny Any. Object associated with key *KeyAny* within dictionary *AnyDict*.

CharInt Integer. Integer value of character from index *IndexInt* within string *AnyStr*.

Comments

In the first variant, the operator recovers the object at index *IndexInt* within the array *AnyArray*. Array indexes range from zero to N-1, where N is the number of elements in the array. The operator pushes a duplicate of the desired object onto the operand stack. For composite objects, the duplicate object shares its data with the original object in the array.

In the second variant, the operator recovers the object associated with *KeyAny* within the dictionary *AnyDict*. The operator pushes a duplicate of the desired object onto the operand stack. For composite objects, the duplicate object shares its data with the original object in the dictionary.

In the third variant, the operator recovers the character at index *IndexInt* within the string *AnyStr*. String character indexes range from zero to N-1, where N is the number of characters in the string. The operator pushes the integer value of the desired object onto the operand stack. PAL uses ASCII encoding for characters within strings.

Hints

The get operator will generate an error if *KeyAny* does not exist within *AnyDict*. If the programmer does not know whether or not *KeyAny* will exist within *AnyDict*, the programmer should use the known operator to test for *KeyAny*.

getinterval

Description

Recover a range of data from an array or string object.

Usage

AnyArray IndexInt LengthInt **getinterval** *SubArray*
AnyStr IndexInt LengthInt **getinterval** *SubStr*

AnyArray Array. Array object containing desired array sub-range to recover.

AnyStr String. String object containing desired string sub-range to recover.

IndexInt Integer. When used with *AnyArray*, index of first data object in sub-range within array. Arrays begin with index zero. When used with *AnyStr*, index of first character in sub-range within string. Strings begin with index zero.

LengthInt Integer. When used with *AnyArray*, number of data objects in sub-range within array. When used with *AnyStr*, number of character in sub-range within string.

SubArray Array. Array containing objects from the *AnyArray* sub-range specified.

SubStr Any. String containing characters from the *AnyStr* sub-range specified.

Comments

In the first variant, the operator recovers the *LengthInt* objects starting at index *IndexInt* within the array *AnyArray*. Array indexes range from zero to N-1, where N is the number of elements in the array. The operator creates the new array *SubArray* from duplicates of the *AnyArray* objects. For composite objects, the duplicate objects share their data with the original objects in the *AnyArray*.

In the second variant, the operator recovers the *LengthInt* characters starting at index *IndexInt* within the string *AnyStr*. String character indexes range from zero to N-1, where N is the number of characters in the string. The operator creates the new string *SubStr* from the characters copied from *AnyArray.*

For both variants, the specified sub-range may exceed the range of either *AnyArray* or *AnyStr*. The intepreter will automatically adjust the requested length to the limits of the source object.

The programmer may also specify a negative value for *LengthInt*. A negative *LengthInt* value instructs the interpreter to index into *AnyArray* or *AnyStr* starting from the end. Therefore, the last object in *AnyArray*, or the last character in *AnyStr* will have an *IndexInt* value of zero. As a result, "[0 1 2 3 4 5] 1 -2 getinterval" will return "[3 4]." For strings, "(ABCDE) 2 -3 getinterval" will return "(ABC)."

Hints

PAL will automatically adjust *LengthInt* for the remaining number of objects left in *AnyArra*y, or characters left in *AnyStr*. By specifying a *LengthInt* equal to or greater than the length of *AnyArray* or *AnyStr*, the programmer can recover the entire end of the array or string. For

example, the following sequences will return all objects or characters starting at index three through the end of MyArray or MyString.

```
MyArray dup 3 exch length getinterval
MyString dup 3 exch length getinterval
```

PAL provides support for negative *LengthInt* values to allow the programmer to easily access the end of an array or string. The following examples return the last five objects in MyArray or the last five characters of MyString.

```
MyArray 0 -5 getinterval
MyString 0 -5 getinterval
```

globaldict

Description

Pushes the global dictionary, globaldict, onto the top of the operand stack.

Usage

globaldict *GlobalDict*

GlobalDict Dictionary. Global dictionary, globaldict, from dictionary stack.

Comments

This operator was introduced for use in future PAL versions. The operator serves no functional purpose at this time. The userdict discussion includes information regarding globaldict.

gt

Description

Determine whether the first object is greater than the second object.

Usage

Any1Num Any2Num **gt** *Bool*
Any1Text Any2Text **gt** *Bool*

Any1Num Integer or fixed-point. First numeric object to compare.

Any2Num Integer or fixed-point. Second numeric object to compare.

Any1Text String or name. First text object to compare.

Any2Text String or name. Second text object to compare.

Bool Boolean. A value of true indicates the first object exceeds the second object. A value of false indicates the second object meets or exceeds the first object.

Comments

In the first variant, the operator determines whether or not the first numeric parameter is greater than the second numeric parameter. The operator will accept integer and fixed-point objects for either parameter.

In the second variant, the operator determines whether or not the first text parameter is greater than the second text parameter. The operator will accept string or name objects for either parameter. PAL compares strings and names using the standard ASCII character sorting sequence including case sensitivity. Therefore, the string (Abc) is less than the string (abc).

idiv

Description

Performs integer division of two numbers, placing the quotient back on the stack.

Usage

DividendInt DivisorInt **idiv** *QuotientInt*

DividendInt Integer. Number which the interpreter will divide by *DivisorInt*.

DivisorInt Integer. Number by which the interpreter will divide *DividendInt*.

QuotientInt Integer. Result of integer division operation.

Comments

On most printers, PAL can perform integer division significantly faster than fixed-point division. The speed difference becomes important in procedures which the interpreter must execute repeated.

Integer division discards any fractional portion of the quotient. Therefore, the operation "5 2 idiv" produces the quotient "2".

if

Description

Conditionally executes a procedure based upon a boolean value.

Usage

AnyBool TrueProc **if**

AnyBool Boolean. Value which determines whether or not PAL executes *TrueProc*. A value of true instructs the interpreter to execute *TrueProc*. A value of false instructs PAL to not execute *TrueProc*.

TrueProc Procedure. The procedure for PAL to execute given a value of true for *AnyBool*.

Comments

This operator provides the PAL programmer with the ability to optionally execute a given procedure. Typically the boolean value *AnyBool* would result from the execution of a comparison operator like eq or gt.

The if operator removes both parameters from the stack before optionally invoking the specified procedure. The if operator does not place any results onto the stack. However, the procedure may place one or more results onto the stack if desired.

Hints

The following three PAL sequences perform entirely different functions.

```
1: 1 1 eq MyProc if
2: 1 1 eq /MyProc if
3: 1 1 eq {MyProc} if
```

The first example instructs PAL to execute the procedure MyProc *before* PAL executes the operator if. As a result, the if operator will generate an error unless MyProc places a procedure object onto the stack before terminating.

In the second example, when the if operator executes, it will encounter a boolean value (true) followed by a literal name (/MyProc) on the stack. Since if expects a boolean value followed by a procedure object, this will produce an error.

The third example shows the proper approach to conditionally execute the procedure MyProc. The if operator will execute the procedure "{MyProc}". The procedure, in turn, executes the procedure MyProc.

The specified procedure may also do more than just execute a saved procedure. The following example sorts the values of the variables High and Low to correspond with their names.

```
High Low lt {/High Low /Low High def def} if
```

ifelse

Description

Conditionally executes one of two procedures based upon a boolean value.

Usage

AnyBool TrueProc FalseProc **`ifelse`**

AnyBool Boolean. Value which determines whether or not PAL executes *TrueProc* or *FalseProc*. A value of true instructs the interpreter to execute *TrueProc*. A value of false instructs PAL to execute *FalseProc*.

TrueProc Procedure. The procedure for PAL to execute given a value of true for *AnyBool*.

FalseProc Procedure. The procedure for PAL to execute given a value of false for *AnyBool*.

Comments

This operator provides the PAL programmer with the ability to execute one of two procedures. Typically the boolean value *AnyBool* would result from the execution of a comparison operator like eq or gt.

The ifelse operator removes all three parameters from the stack before invoking the selected procedure. The ifelse operator does not place any results onto the stack. However, the executed procedure may place one or more results onto the stack if desired.

Hints

The if operator discussion includes hints which also apply to the ifelse operator.

imagemask

Description

Draws a rasterized image.

Usage

WNum HNum PolBool TmArray SrcProc **`imagemask`**

WNum Integer. Width of source raster image in pixels.

HNum Integer. Height of source raster image in pixels.

PolBool Boolean. true indicates positive image polarity. false indicates negative image polarity. For positive image polarity, image bits with a value of one indicate black pixels. Image bits with a value of zero indicate white pixels. For negative image polarity, image bits with a value of zero indicate black pixels. Image bits with a value of one indicate white pixels.

TmArray Array. Specifies the source raster image's transformation matrix. At this time, the programmer should only specify this array as follows.

`[`*WNum* **`0 0 -`***HNum* **`0`** *HNum***`]`**

WNum Integer. Same as the width of the source raster image in pixels.

`-`*HNum* Integer. Negative height of the source raster image in pixels.

HNum Integer. Same as the height of the source raster image in pixels.

SrcProc Procedure. Procedure which supplies the source data for the raster image. PAL continues to execute this procedure until the procedure returns sufficient data to complete the raster image. The *WNum* and *HNum* parameters establish the amount of data required to complete the raster image.

Comments

The interpreter relies upon the specified procedure to supply the data for the raster image. Once the interpreter executes the procedure, the procedure must place a string object onto the stack before terminating. Upon termination of the procedure, the interpreter recovers the string object from the stack and appends the data to any previous string objects returned by the procedure.

The interpreter will continue to execute the procedure until the procedure has returned sufficient data to complete the raster image. PAL determines the amount of data necessary for the raster image from the *WNum* and *HNum* parameters.

PAL has a standardized format for representing the raster data within the strings return by the procedure. A raster image consists of an array of bits. Each bit specifies the color of a single pixel. As discussed above, the *PolBool* parameter specifies the association between bit values and colors.

The first bit of the first byte returned by the procedure specifies the color of the top left pixel of the raster image. The next bit specifies the color of the next pixel to the right. Each successive bit

specifies the color of each successive pixel to the right. This process continues for *WNum* bits and pixels. Bit number *WNum* specifies the color of the right-most pixel on the top-most line.

Bit number *WNum*+1 specifies the color of the left-most pixel on the second line from the top. Unlike most raster image formats, the PAL raster image format does not include any unused bits between the last bit of one line and the first bit of the next line.

PAL treats the most significant bit of each byte as the first bit of the byte. PAL treats the least significant bit of each byte as the eighth bit of the byte.

The following table illustrates the bit numbers for each pixel in a 10 by 10 pixel raster image.

00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**
10	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
20	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**
30	**31**	**32**	**33**	**34**	**35**	**36**	**37**	**38**	**39**
40	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**
50	**51**	**52**	**53**	**54**	**55**	**56**	**57**	**58**	**59**
60	**61**	**62**	**63**	**64**	**65**	**66**	**67**	**68**	**69**
70	**71**	**72**	**73**	**74**	**75**	**76**	**77**	**78**	**79**
80	**81**	**82**	**83**	**84**	**85**	**86**	**87**	**88**	**89**
90	**91**	**92**	**93**	**94**	**95**	**96**	**97**	**98**	**99**

The PAL raster format requires the programmer to group the bits for the above image into bytes, as follows.

	Bit 7	6	5	4	3	2	1	0
Byte 0	00	01	02	03	04	05	06	07
Byte 1	08	09	10	11	12	13	14	15
Byte 2	16	17	18	19	20	21	22	23
Byte 3	24	25	26	27	28	29	30	31
Byte 4	32	33	34	35	36	37	38	39
Byte 5	40	41	42	43	44	45	46	47
Byte 6	48	49	50	51	52	53	54	55
Byte 7	56	57	58	59	60	61	62	63
Byte 8	64	65	66	67	68	69	70	71
Byte 9	72	73	74	75	76	77	78	79
Byte 10	80	81	82	83	84	85	86	87
Byte 11	88	89	90	91	92	93	94	95
Byte 12	96	97	98	99	xx	xx	xx	xx

Normally, the procedure specified for the imagemask operator will simply contain a single hexadecimal string containing the raster image data. In the case of the above image, the entire imagemask operation would appear as follows.

```
10 10 true [10 0 0 -10 0 10]
  {<0C 0C C4 09 4A 80 64 94 C9 02 33 03 00>}
  imagemask
```

Since a 10 by 10 pixel image only requires 100 bits of data, PAL ignores all bits returns by the procedure following bit 99. Therefore, as shown by the "xx" bits in the byte table above, PAL will ignore the last four bits of the last byte.

The imagemask operator always draws the raster image with the lower left corner of the image positioned at the user coordinate system origin. The current point, as established by moveto and other operators, has no effect upon the placement of an imagemask raster image. The programmer must use the translate operator to position the raster image on the page.

The scaling and rotation of the user coordinate system also affects the image. The interpreter rotates and scales the image so that the image's lower left corner will appear at coordinate 0,0 and the upper right corner will appear at coordinate 1,1. Therefore, the programmer can use the scale and rotate operators to influence the final image.

Hints

The programmer should consider applying the desired translate, rotate, and scale operations immediately before the imagemask operation. Then, immediately following the imagemask operation, the programmer should reverse any necessary transformations.

For example, the following code will draw the above "smiling face" with the lower left corner at 0.5",0.75". The code will scale the image to 2" wide by 1" tall.

```
initmatrix
36 54 translate
144 72 scale

10 10 true [10 0 0 -10 0 10]
  {<0C 0C C4 09 4A 80 64 94 C8 02 33 03 00>}
  imagemask

1 144 div 1 72 div scale
-36 -54 translate
```

_imp

Description

Performs a logical or bit-wise implication operation on two boolean or integer values.

Usage

Any1Bool Any2Bool **_imp** *ImpBool*
Any1Int Any2Int **_imp** *ImpInt*

Any1Bool Boolean. First operand for the logical implication operation.

Any2Bool Boolean. Second operator for the logical implication operation.

ImpBool Boolean. Result of the logical implication operation.

Any1Int Integer. First operand for the bit-wise implication operation.

Any2Int Integer. Second operand for the bit-wise implication operation.

ImpInt Integer. Result of the bit-wise implication operation.

Comments

The following table lists the results of performing the logical implication operation on two boolean values.

		Any1Bool	
		false	true
Any2Bool	false	true	false
	true	true	true

The following table lists the results for each bit position when performing the bit-wise implication operation on two integer values.

		Any1Int	
		0	1
Any2Int	0	1	0
	1	1	1

index

Description

Recovers an object from a lower level of the operand stack.

Usage

NAny..0Any IndexInt **index** *NAny..0Any IndexedAny*

NAny..0Any Any. All objects on the operand stack.

IndexInt Integer. Index of desired object on operand stack. The top-most object, prior to pushing *IndexInt*, has an index of zero. The next lower object has an index of one.

IndexedAny Any. Requested object from operand stack.

Comments

This operator functions in a manner similar to the dup operator. However, index duplicates any object on the operand stack. For composite objects, the original and duplicate objects will share the same composite data.

initgraphics

Description

Restores default values to various settings within the current graphics state.

Usage

```
initgraphics
```

Comments

PAL restores the following graphics state settings to the indicated defaults.

Graphics State Setting	Default Value
Transformation Matrix	Device Default (Normally Points)
Path	Empty
Current Point	Undefined
Line Width	One User Coordinate System Unit
Line Cap Style	Butt End Cap

initmatrix

Description

Reset user coordinate system to PAL default coordinate system.

Usage

```
initmatrix
```

Comments

The PAL default coordinate system locates the origin at the bottom left corner of the page, a scaling factor of 1/72", and no rotation.

known

Description

Determines whether or not a given key exists within a given dictionary.

Usage

AnyDict KeyAny **known** *Bool*

AnyDict	Dictionary. Dictionary object to search for *KeyAny*.
KeyAny	Any. Key object for which to search in *AnyDict.*
Bool	Boolean. A value of true indicates that PAL found *KeyAny* within *AnyDict*. A value of false indicates that PAL did not find *KeyAny* within *AnyDict*.

Comments

The get operator generates an error if the specified key object does not exist within the specified dictionary. If the programmer does not already know that a key exists within a given dictionary, the programmer should test for the key using the known operator before using the get operator.

le

Description

Determine whether the first object is less than or equal to the second object.

Usage

Any1Num Any2Num **le** *Bool*
Any1Text Any2Text **le** *Bool*

Any1Num Integer or fixed-point. First numeric object to compare.

Any2Num Integer or fixed-point. Second numeric object to compare.

Any1Text String or name. First text object to compare.

Any2Text String or name. Second text object to compare.

Bool Boolean. A value of true indicates the first object does not exceed the second object. A value of false indicates the second object exceeds the first object.

Comments

In the first variant, the operator determines whether or not the first numeric parameter is less than or equal to the second numeric parameter. The operator will accept integer and fixed-point objects for either parameter.

In the second variant, the operator determines whether or not the first text parameter is less than or equal to the second text parameter. The operator will accept string or name objects for either parameter. PAL compares strings and names using the standard ASCII character sorting sequence including case sensitivity. Therefore, the string (Abc) is less than the string (abc).

length

Description

Returns the size of the supplied parameter.

Usage

```
AnyArray   length   ElementsInt
 AnyDict   length   PairsInt
  AnyStr   length   CharsInt
```

AnyArray	Array. Array object of which to return its length.
ElementsInt	Integer. Number of objects in array.
AnyDict	Dictionary. Dictionary object of which to return its length.
PairsInt	Integer. Number of key + value pairs in dictionary.
AnyStr	String. String object of which to return its length.
CharsInt	Integer. Number of characters in string.

Comments

In the first variant, the operator returns the number of objects contained in the array. In the second variant, the operator returns the number of key + value pairs contained in the dictionary. Therefore, a dictionary will actually contain twice as many objects as the count returned by length. The third variant returns the number of characters contained in the string.

lineto

Description

Appends a line to the current path which extends from the current point to the specified point.

Usage

XNum YNum **`lineto`**

XNum Integer or fixed-point. Specifies the X component of the user coordinate to which to extend the line.

YNum Integer or fixed-point. Specifies the Y component of the user coordinate to which to extend the line.

Comments

The lineto operator does not actually draw a line on the current page. Instead, PAL adds the line to a drawing *path* which it keeps in memory. As the programmer specifies additional drawing operations, PAL continues to append these operations to the end of the drawing path. Once the programmer has specified the entire path for PAL to draw, the programmer uses the stroke operator to instruct PAL to actually *stroke* the path onto the page. PAL also provides the setlinewidth and setlinecap operators which influence the stroke operator.

In general, the programmer should not use lineto to finish a path around a closed object such as a square. For a square, the drawing sequence should use lineto to draw the first three sides of the square, but closepath for the final side. closepath instructs PAL to join the end of the closepath line to the first point of the path. This allows PAL to smooth the transition from the last line to the first line during the stroke operation.

After appending the lineto operation to the path, PAL updates the current point in the graphics state to the specified *XNum*, *YNum* coordinate. This allows subsequent drawing orders to automatically continue from the end of the line.

PAL currently restricts line drawing to horizontal and vertical lines. Therefore, in order to receive expected results, either the *XNum* or *YNum* parameter should match the current point's X or Y position.

Hints

The rlineto operator provides an easier means for drawing lines of a given length.

_localtime

Description

Returns the current time of day and date if available.

Usage

_localtime *TimeArray*

TimeArray Array.

[*AvailBool TotalInt YearInt MonthInt DayInt HourInt MinInt SecInt Sec100Int DOYInt DOWInt ZoneFxp SaveInt***]**

AvailBool Boolean. true if printer knows the time. false if printer does not know the time.

TotalInt Integer. Total number of seconds elapsed since 00:00:00 on January 1, 1970.

YearInt Integer. Current year. 1990, 1991, etc.

MonthInt Integer. Current month of the current year. 1 to 12.

DayInt Integer. Current day of the current month. 1 to 31.

HourInt Integer. Current hour of the current day. 0 to 23.

MinInt Integer. Current minute of the current hour. 0 to 59.

SecInt Integer. Current second of the current minute. 0 to 59.

Sec100Int Integer. Current 1/100 second of the current second. 0 to 99.

DOYInt Integer. Current day of the current year. 1 to 366.

DOWInt Integer. Current day of the current week. 1 to 7.

ZoneFxp Fixed-point. Local time zone if known by the printer. 0.0 to 23.0 or -1.0. Provides the relationship between the local time and Greenwich mean time (GMT). Subtracting *ZoneInt* from the local time provides GMT. PAL will return -1.0 if the printer does not know the local time zone.

SaveInt Integer. One if time based on daylight savings time. Zero if standard time. -1 if printer does not know.

Comments

Nearly all PAL printers have the ability to maintain the current time and date so long as they remain powered-on. Only some PAL printers have the ability to maintain the correct time and date

while powered-off. The programmer should check the printer's documentation for information concerning date and time maintenance.

At some point, all PAL printers require an operator to set the current date and time. Printers which can maintain the correct date and time while powered-off only require the operator to set the time once, or once following any time changes. Printers which cannot maintain the correct time when powered-off need the operator to set the time following power-on.

In either case, if the printer never receives the correct time from the operator, the _localtime operator will set *AvailBool* to false in the returned array. This indicates that the printer does not know the current time or date.

In addition, even if the printer knows the correct time and date, the printer may not know the local time's relationship to Greenwich mean time (GMT). The printer may also not know whether or not the local time zone has daylight savings time active. Whether or not the printer knows these two settings usually depends upon whether or not the operator told the printer. PAL will return -1 for either or both of these values if the printer does not contain the information.

The *ZoneInt* and *SaveInt* values provide the extended time logging information often required by wide area network systems operating across multiple time zones.

Special Considerations

Some printers may normally indicate the correct daylight savings time status at all times, excluding two special hours during the year. At 2:00:00 AM on the last Sunday in October, daylight savings time ends and standard time begins. During this transition, clocks are updated from 1:59:59 AM to 1:00:00 AM. Therefore, a time reading bwtween 1:00:00 AM and 1:59:59 AM on this day may indicates either 1-2 AM daylight savings time or 1-2 AM standard time. During this two hour period of time, some printers may automatically indicate "unknown" (-1) for *SaveInt*.

loop

Description

Repetitively executes the specified procedure until the interpreter encounters an exit operator.

Usage

AnyProc **loop**

AnyProc Procedure. The procedure which the interpreter will repetitively execute.

Comments

PAL first removes the supplied procedure from the stack. The interpreter then continuously executes the procedure until the interpreter encounters an exit operator. The loop operator does not place any objects on the operand stack. However, the procedure may place objects on the stack if desired.

Hints

The following three PAL sequences perform entirely different functions.

```
1: MyProc loop
2: /MyProc loop
3: {MyProc} loop
```

The first example instructs PAL to execute the procedure MyProc *before* PAL executes the operator loop. As a result, the loop operator will generate an error unless MyProc places a procedure object onto the stack before terminating.

In the second example, when the loop operator executes, it will encounter a literal name (/MyProc) on the stack. Since loop expects a procedure object, this will produce an error.

The third example shows the proper approach to repetitively execute the procedure MyProc. The loop operator will repetitively execute the procedure "{MyProc}". The procedure, in turn, executes the procedure MyProc. Since the "{MyProc}" procedure does not include an exit operator, the MyProc procedure must contain an exit operator. Otherwise, the loop will continue to execute forever.

The specified procedure may also do more than just execute a saved procedure. The following example sums a series of numbers on the operand stack until it encounters a zero.

```
dup 0 ne {{exch dup 0 eq {pop exit} if add} loop} if
```

lt

Description

Determine whether the first object is less than the second object.

Usage

Any1Num Any2Num **lt** *Bool*
Any1Text Any2Text **lt** *Bool*

Any1Num Integer or fixed-point. First numeric object to compare.

Any2Num Integer or fixed-point. Second numeric object to compare.

Any1Text String or name. First text object to compare.

Any2Text String or name. Second text object to compare.

Bool Boolean. A value of true indicates the first object does not meet or exceed the second object. A value of false indicates the second object meets or exceeds the first object.

Comments

In the first variant, the operator determines whether or not the first numeric parameter is less than the second numeric parameter. The operator will accept integer and fixed-point objects for either parameter.

In the second variant, the operator determines whether or not the first text parameter is less than the second text parameter. The operator will accept string or name objects for either parameter. PAL compares strings and names using the standard ASCII character sorting sequence including case sensitivity. Therefore, the string (Abc) is less than the string (abc).

_ltrim

Description

Eliminate undesired characters from left (leading) edge of a string.

Usage

AnyStr SetStr **_ltrim** *TrimmedStr*

AnyStr	String. String possibly containing character to eliminate.
SetStr	String. Set of undesired characters. An empty string "()" instructs the interpreter to trim whitespace characters.

Comments

PAL creates the new string *TrimmedStr* by copying the contents of *AnyStr*. As PAL copies the characters from left to right, PAL compares each character to the list of characters contained in *SetStr*. If PAL finds the character in *SetStr* it does not copy the character to *TrimmedStr*. As soon as PAL finds a character from *AnyStr* which does not match a character in *SetStr*, PAL then copies that character, and all remaining *AnyStr* characters to *TrimmedStr*.

Specifying an empty string "()" for *SetStr* instructs PAL to trim all whitespace characters from the left edge of *AnyStr*. PAL treats all characters with a decimal value of 32 and below as whitespace characters. This includes ASCII spaces, tabs, carriage returns, line feeds, as well as all other standard ASCII non-printable control characters.

makefont

Description

Modifies a font dictionary to scale the characters with optional independent X and Y factors and optional mirroring.

Usage

AnyFontDict TmArray **`makefont`** *TmFontDict*

AnyFontDict Dictionary. Font dictionary returned by the findfont operator.

TmArray Array. Transformation matrix to apply to scale the font's characters to the desired X and Y dimensions. PAL modifies *AnyFontDict*. It does not create a new dictionary.

Comments

Most users will find the scalefont simpler to use than the makefont operator. The makefont operator requires additional complexity in relation to the scalefont operator in order to provide additional capabilities. Users who simply wish to scale a font to a desired height should use the scalefont operator.

The font dictionary returned by the findfont operator contains information designed to scale the characters of the font to one user coordinate tall. The makefont operator modifies the font dictionary to draw characters of any desired height and width. makefont also allows for drawing characters as mirror images.

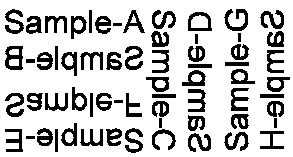
The font dictionary contains only relative character scaling information. PAL combines the font's scaling information with the current user scaling factor to determine the true size of the characters. PAL combines this information when it draws the characters.

Since fonts default to one user coordinate tall, and the user coordinate system defaults to one point scaling, each character defaults to a height of one point. However, changing the user coordinate scaling factor will accordingly change the default character height. For example, changing the user coordinate system to inches changes the default character height to one inch.

The makefont operator allows the programmer to compensate for the current user coordinate system scaling as well as establish any desired height and width for the characters. Under the default coordinate scaling of points, the "*AnyFontDict* [12 0 0 12 0 0] makefont" operation will configure the font information to draw 12 point characters. However, the same makefont operation with a user coordinate system based on inches would produce 12 inch tall characters. Therefore, with a user coordinate system based on inches, "*AnyFontDict* [12 72 div 0 0 12 72 div 0 0] makefont" will configure the font information to draw 12 point characters.

Although the scalefont operator requires less information, the makefont operator allows the user to specify different scaling factors for the X and Y dimensions of the characters. The user can also rotate the characters independent of rotating the coordinate system, as well as mirror the characters. The following table summarizes the use of *TmArray* to produce various standard character transformations.

Character's Rotation	Mirror Character's X	Mirror Character's Y	*TmArray*	Sample (Below)
0	No	No	**[***X* **0 0** *Y* **0 0]**	Sample-A
0	Yes	No	**[-***X* **0 0** *Y* **0 0]**	Sample-B
0	No	Yes	Same as 180, Yes, No.	Sample-F
0	Yes	Yes	Same as 180, No, No.	Sample-E
90	No	No	**[0 -***X* *Y* **0 0 0]**	Sample-C
90	Yes	No	**[0** *X* *Y* **0 0 0]**	Sample-D
90	No	Yes	Same as 270, Yes, No.	Sample-H
90	Yes	Yes	Same as 270, No, No.	Sample-G
180	No	No	**[-***X* **0 0 -***Y* **0 0]**	Sample-E
180	Yes	No	**[***X* **0 0 -***Y* **0 0]**	Sample-F
180	No	Yes	Same as 0, Yes, No.	Sample-B
180	Yes	Yes	Same as 0, No, No.	Sample-A
270	No	No	**[0** *X* **-***Y* **0 0 0]**	Sample-G
270	Yes	No	**[0 -***X* **-***Y* **0 0 0]**	Sample-H
270	No	Yes	Same as 90, Yes, No.	Sample-D
270	Yes	Yes	Same as 90, No, No.	Sample-C



In the table, *X* and *Y* indicate the scale factors along the character's dimension. The *X* scale factor controls the width of the character, and *Y* controls the height. For nearly all fonts, characters scaled to a given height of *Y* will never be exactly *Y* units tall. Each character will be the appropriate size as determined by the artistic design of the font in proportion to the requested overall font height.

For the same reasons, specifying a given *X* width for a font only requests the rendering of each character at its appropriate artistic width equivalent to the font rendered at a height of *X* units. The characters of most fonts are typically much narrower than they are tall. As a result, characters will seldom actually approach *X* units wide.

Based on this information, a *TmArray* of "[12 0 0 12 0 0]" will produce a 12 unit font. This would be identical to using a *ScaleNum* value of 12 with the scalefont operator. To produce a double-wide variation of the same font, simply specify "[24 0 0 12 0 0]". The only difference being the doubling of the *X* scaling value. To produce a double tall variation, simply double the *Y* scaling value by specifying "[12 0 0 24 0 0]".

Hints

PAL does not automatically scale all the characters of a font in response to the makefont operator. Since character scaling takes time, PAL waits to scale each character until it needs to draw the character onto a page. PAL also rotates the characters at this time to match the orientation at which it must draw the characters.

Once PAL scales and rotates a character, it places the character image into a *character cache*. This allows PAL to use the already scaled character image if PAL must draw the character again.

mark

Description

Pushes a special mark object onto the operand stack.

Usage

`mark` *Mark*

Mark Mark. Stack position marking object.

Comments

PAL provides a special data object type knows as a *mark* object. Numerous PAL operators rely upon the use of a mark object to mark a position on the operand stack.

_measurepage

Description

Instructs PAL to measure the size of the current media if possible.

Usage

LimitsArray PagesInt **_measurepage** *SizeArray*

LimitsArray Array. Specifies the maximum dimensions of the media.

[*WidthMaxNum HeightMaxNum InterPageMaxNum***]**

WidthMaxNum Integer or fixed-point. Specifies the maximum width of the media in points (1/72"). This value does not use the user coordinate system.

HeightMaxNum Integer or fixed-point. Specifies the maximum height of the media in points (1/72"). This value does not use the user coordinate system.

InterPageMaxNum
Integer or fixed-point. Specifies the maximum size of the gap between pages in points (1/72"). This value typically only applies to continuous media printers. This value does not use the user coordinate system.

PagesInt Integer. Specifies the number of pages across which PAL will average the media measurements.

SizeArray Array. Gives the dimensions of the installed media.

[*WidthNum HeightNum InterPageNum***]**

WidthNum Integer or fixed-point. Specifies the width of the media, as measured by the mechanism, in points (1/72"). This value does not use the user coordinate system. PAL will return -1 for this value if the mechanism cannot perform the required measurement.

HeightNum Integer or fixed-point. Specifies the height of the media, as measured by the mechanism, in points (1/72"). This value does not use the user coordinate system. PAL will return -1 for this value if the mechanism cannot perform the required measurement.

InterPageNum Integer or fixed-point. Specifies the distance between pages, as measured by the mechanism, in points (1/72"). This value does not use the user coordinate system. This value typically only applies to continuous media printers. PAL will return -1 for this value if the mechanism cannot perform the required measurement, or the measurement does not apply to the installed media.

Comments

Not all PAL printers can automatically measure the media, and those which can may not perform all the measurements indicated. In printers that do not measure the media, the _measurepage operator will simply return the current page settings.

Many printers must consume a certain amount of the media in order to measure it. In addition, other PAL settings can influence the printer's ability to correctly measure the media. As a result, the *LimitArray* and *PagesInt* parameters provide a limitation on how much media the printer should consume. Not all printers will use *PagesInt* or all the entries in *LimitArray*. Some printers can establish their own limits. On printers that do not actually measure the page but simply return values, the *PagesInt and LimitArray* are ignored.

Pressure fed continuous media printers provide the primary reason for the existence of this operator. These types of printers use special sensors to detect the beginning and end of each page. Since these printers use pressure rollers to advance the media, the media can slip very slightly in relation to the printing mechanism. This slipping can result in some pages appearing slightly longer or shorter in relation to other pages as the pages move past the sensors in the printer.

These printers must continuously adjust the feeding of the media in order to compensate for this slipping. The printer calculates these adjustments based on the anticipated size of the media versus the size measured by the sensors. In order to maintain accurate positioning of the media, the printer must receive as accurate a media measurement as possible.

Unfortunately, varying media thickness and other variables can also result in pressure feed printers not feeding the media in a consistent manner. As a result, different media types can introduce different amounts of error.

By asking the mechanism itself to measure the media, the mechanism will automatically include any error introduced by the media into the measurements. This will result in a measurement which closely matches the anticipated operation of the media within the mechanism.

On pressure fed mechanisms, the programmer should specify a *PagesInt* value of around five or more to ensure a reasonable level of consistency from one measurement to the next. Normal measurement fluctuation on a pressure fed mechanism can often range as high as three points. Excessively high fluctuations probably indicate sensor or other adjustment problems.

mod

Description

Perform integer division and return the modulo (remainder).

Usage

DividendInt DivisorInt **mod** *RemainderInt*

DividendInt Integer. Value which PAL will divide by *DivisorInt*.

DivisorInt Integer. Value which PAL will divide into *DividendInt*.

RemainderInt Integer. Remainder following the division.

Comments

This operator calculates the remainder following an integer division. For example, "5 2 mod" produces the result "1".

moveto

Description

Establishes the new current point.

Usage

XNum YNum **moveto**

XNum Integer or fixed-point. New current X coordinate. Specified in user coordinates.

YNum Integer or fixed-point. New current Y coordinate. Specified in user coordinates.

Comments

Most PAL drawing operators use the current point as the location at which to draw. In most cases, the current point establishes the bottom left corner of any new object drawn.

The moveto operator also has the affect of ending any active sub-path in the graphics state and starting a new sub-path. Depending upon the desired result, the programmer may wish to specify closepath before a moveto in order to close the active sub-path. Using moveto without closepath creates an open sub-path.

Closing a sub-path instructs PAL to smooth the connection between the first and last lines of the path. PAL assumes that the start and end points of an open path either do not meet or the programmer does not wish PAL to smooth the connection.

mul

Description

Multiplies two numbers and returns the product.

Usage

Any1Num Any2Num **mul** *ProductNum*

Any1Num Integer or fixed-point. First number to multiply.

Any2Num Integer or fixed-point. Second number to multiply.

SumNum Integer or fixed-point. Integer if *Any1Num* and *Any2Num* are both integer, otherwise fixed-point. Product of *Any1Num* and *Any2Num*.

Comments

The mul operator pops the top two objects from operand stack, multiplies them together, and pushes the result back onto the operand stack. The interpreter must find two numeric objects on the top of the stack or a typecheck error will result.

If the stack contains two integer objects, the interpreter will perform integer multiplication and push an integer result onto the stack. The interpreter will perform fixed point multiplication and push a fixed point result if the stack contains a fixed point object as either operand.

ne

Description

Compare two objects for inequality.

Usage

1Any 2Any **ne** *Bool*

1Any	Any. First object to compare. With the exception of integer, fixed-point, string, and name objects, *1Any* must have the same object type as *2Any*. The operator will compare integer and fixed-point objects in any combination. The operator will also compare any combination of string and name objects.
2Any	Any. Second object to compare.
Bool	Boolean. A value of true indicates inequality. A value of false indicates equality.

Comments

PAL will compare for inequality any two objects of the same object type. PAL will also compare any combination of integer and fixed-point objects, as well as any combination of string and name objects.

PAL compares strings and names using the standard ASCII character sorting sequence including case sensitivity. Therefore, the string (abc) does not match the string (ABC).

For composite objects, only if the two objects reference the exact same composite data does PAL consider the objects equal. Therefore, the PAL sequence "[1 2 3] [1 2 3] ne" produces the result "true". The sequence "[1 2 3] dup ne" produces the result "false".

In the first case, the sequence creates two unique arrays which happen to contain the same data. Since the array objects reference different data within the printer's memory, the objects do not meet PAL's condition for equality.

In the second case, the sequence creates a single array and a second reference to the same array data. Since the array objects reference the same data with the printer's memory, the objects meet PAL's condition for equality.

Hints

PAL uses the same conditions for equality and inequality for the eq and ne operators as it does when comparing key values in dictionaries.

neg

Description

Returns the twos-complement of any number.

Usage

AnyNum **neg** *NegNum*

AnyNum	Integer or fixed-point. Number from which to return twos-complement of its value.
NegNum	Integer or fixed-point. Negative value of *AnyNum*. Same object type as *AnyNum*.

Comments

The neg operator pops the top object from operand stack, subtracts the value from zero, and pushes the result onto the operand stack. The result's type will match the original value's type.

newpath

Description

Discards any path information in the current graphics state.

Usage

```
newpath
```

Comments

PAL operators which perform drawing operations using the current path also automatically perform a newpath operation. However, PAL provides this operator to allow PAL procedures to explicitly discard the current drawing path.

not

Description

Performs a logical complement or bit-wise ones-complement operation on a boolean or integer value.

Usage

AnyBool **not** *NotBool*
AnyInt **not** *NotInt*

AnyBool Boolean. Operand for the logical complement operation.

NotBool Boolean. Result of the logical complement operation.

AnyInt Integer. Operand for the bit-wise ones-complement operation.

NotInt Integer. Result of the bit-wise ones-complement operation.

Comments

The following table lists the results of performing the logical complement operation a boolean value.

AnyBool	false	true
Result	true	false

The following table lists the results for each bit position when performing the bit-wise ones-complement operation on an integer value.

AnyInt	0	1
Result	1	0

null

Description

Pushes a null object onto the operand stack.

Usage

`null` *Null*

Null Null. Null object.

Comments

PAL provides the null object type to use as "filler" objects in various situations. For example, when the array operator creates a new array object, it fills the array with null objects.

or

Description

Performs a logical or bit-wise *or* operation on two boolean or integer values.

Usage

Any1Bool Any2Bool **or** *OrBool*
Any1Int Any2Int **or** *OrInt*

Any1Bool Boolean. First operand for the logical *or* operation.

Any2Bool Boolean. Second operator for the logical *or* operation.

OrBool Boolean. Result of the logical *or* operation.

Any1Int Integer. First operand for the bit-wise *or* operation.

Any2Int Integer. Second operand for the bit-wise *or* operation.

OrInt Integer. Result of the bit-wise *or* operation.

Comments

The following table lists the results of performing the logical *or* operation on two boolean values.

		Any1Bool	
		false	true
Any2Bool	false	false	true
	true	true	true

The following table lists the results for each bit position when performing the bit-wise *or* operation on two integer values.

		Any1Int	
		0	1
Any2Int	0	0	1
	1	1	1

_ostimeget

Description

Returns value of operating system clock

Usage

`_ostimeget` *SytemTimeInt*

SystemTimeInt Integer. Value of the operating system clock in increments of 5ms.

Comments

This operator retrieves the value of the operating system clock. The operating system clock is reset to 0 at power up and counts in 5ms increments. It rolls over every 248 days if printer is left on for that long!

_play

Description

Plays a musical sequence.

Usage

ScoreStr **_play**

ScoreStr String. Instructions for PAL's Solo Wave Audio (SWA) generator.

Comments

This operator provides the simplest means of controlling the speaker or other audio source available on many PAL printers. With this operator, the programmer can generate musical sequences to create audio-feedback for a printer operator.

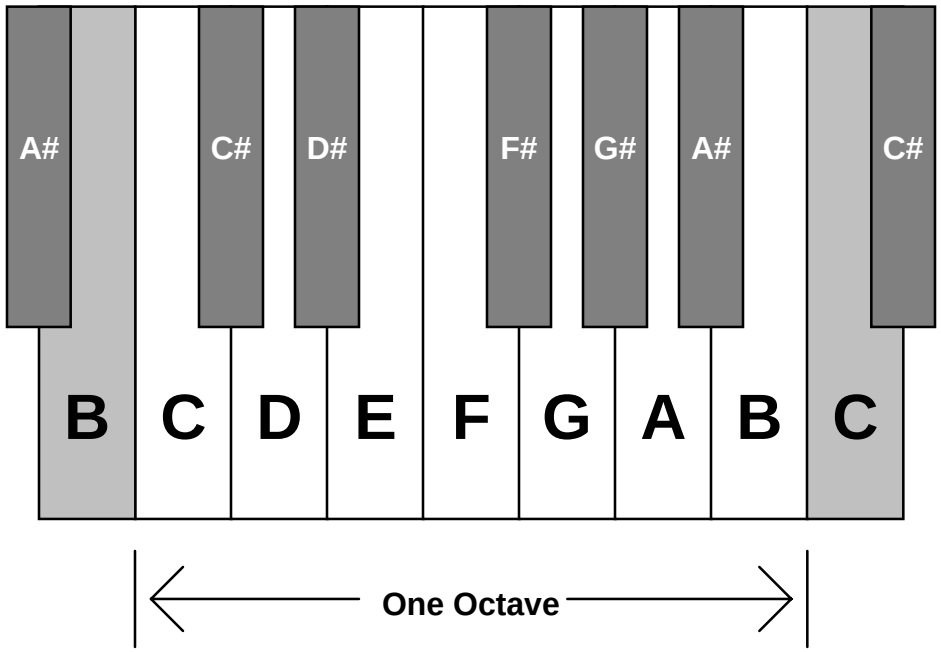
PAL refers to this level of audio control as *Solo Wave Audio* support. "Solo Wave" refers to the fact that the printer must only generate a single pure tone (audio wave) for each requested note.

PAL printers typically provide three levels of audio generation capability in relation to Solo Wave Audio support. Some PAL printers do not contain a speaker or other audio generator. These printers will simply ignore Solo Wave Audio requests generated by the _play operator.

Some PAL printers provide a small buzzer, or possibly a speaker, which can produce an audio tone with only a single frequency. Since these printers can only produce "beeps," they will respond to Solo Wave Audio requests by generating the same "beep" for every note requested. On some of these printers, the Solo Wave Audio sequence may have some influence over the duration of each "beep."

Printers with more extensive Solo Wave Audio support can produce notes at different frequencies. The total range of available notes can vary from printer to printer. If a printer receives a request to play a note outside its supported range, the printer will typically substitute the closest supported note.

PAL supports seven octaves of notes. Each octave contains 12 total notes. This makes a total of 84 notes available for playing. Octaves begin with the note "C" at the low end and range up to note "B" at the high end. The following diagram shows one octave on a piano keyboard.



The *ScoreStr* parameter to the _play operator contains a human readable sequence of simple one and two character instructions. These instructions provide PAL with the necessary information to play both short and long melodies.

The following table lists the one and two character instructions which can reside within *ScoreStr*. PAL treats all instructions as case sensitive. Therefore, the programmer must specify the instructions in upper or lower case exactly as shown in the table.

Instruction	**Operation**
A..**G**	Play the indicated note within the current octave. **A**..**G** indicates any one of the upper case letters from A to G inclusive. As listed below, the user may use the characters **#**, **+**, **-**, and **.** following each note to modify the note's pitch and duration.
L*n*	Set the length of all subsequent notes. *n* specifies the note length as 1/*n*. For example, L1 specifies whole-notes, L2 specifies half-notes, L3 specifies third-notes. *n* may range from one to 64.
ML	Selects "music legato" for all subsequent notes. Music legato plays each note for the entire duration of its length.
MN	Selects "music normal" for all subsequent notes. Music normal plays each note for 7/8 of the note's length. A rest of 1/8 of the note's length will follow each note.
MS	Selects "music staccato" for all subsequent notes. Music staccato plays each note for 3/4 of the note's length. A rest of 1/4 of the note's length will follow each note.

N*n* Plays note *n* out of the 84 possible notes. *n* may range from one to 84, or zero. N36 specifies middle C. N37 specifies middle C-sharp. N38 specifies middle D. N0 specifies a rest note.

o*n* Selects octave *n* out of the seven possible octaves for all subsequent notes. *n* may range from zero to six. o0 selects the lowest octave. o3 selects the middle octave, which contains middle C. o6 selects the highest octave. The user must specify the "o" in lower case.

P*n* Specifies a rest note of length 1/*n*. For example, P1 specifies a whole-note rest, P2 specifies a half-note rest, L3 specifies third-note rest. *n* may range from one to 64.

T*n* Selects the tempo for all subsequent notes. *n* specifies the number of quarter-notes per minute. For example, T120 selects 120 quarter-notes per minute. *n* may range from 32 to 255.

> Selects the next higher octave. If the user has the highest octave selected, PAL will ignore this instruction.

< Selects the next lower octave. If the user has the lowest octave selected, PAL will ignore this instruction.

Sharpens the preceding note. For example, C# selects C-sharp.

+ Sharpens the preceding note. For example, C+ selects C-sharp. Same as #.

- Flattens the preceding note. For example, C- selects C-flat.

. Lengthens the preceding note by 50%. Each subsequent "." will lengthen the note by an additional 50%. For example, "L4C" will play a quarter-note C. "L4C#." will play a 3/8-note C-sharp. "L4E-.." will play a 7/16-note E-flat.

The following example plays the full middle scale in a rising manner at 120 quarter-notes per second using music legato.

```
(T120MLL4o3CC#DD#EFF#GG#AA#B) _play
```

The following example plays a two octave falling scale at 70 half-notes (140 quarter-notes) per second using music staccato.

```
(T140MSL2o4BB-AA-GG-FEE-DD-C<BB-AA-GG-FEE-DD-C) __play
```

Hints

PAL does not provide a mundane "beep" or similar operator. Instead, PAL relies upon any operation equivalent to "(C) _play" to perform this function. The user should experiment with different notes and durations to get the desired effect.

pop

Description

Discards the object on the top of the operand stack.

Usage

Any **pop**

Any Any. Object to discard.

Comments

This operator provides the primary means for discarding unwanted objects from the top of the operand stack.

print

Description

Writes the contents of a string to %stdout.

Usage

AnyStr **`print`**

AnyStr String. Data to write to %stdout.

Comments

PAL writes the string contents to %stdout without any changes or surrounding data.

put

Description

Store data into a composite object.

Usage

AnyArray IndexInt ElementAny **put**
AnyDict KeyAny ValueAny **put**
AnyStr IndexInt CharInt **put**

AnyArray Array. Array object into which to store *ElementAny*.

IndexInt Integer. When used with *AnyArray*, index within array at which to store *ElementAny*. Arrays begin with index zero. When used with *AnyStr*, index within string at which to store *CharInt*. Strings begin with index zero.

ElementAny Any. Object to store at index *IndexInt* within array *AnyArray*.

AnyDict Dictionary. Dictionary object into which to store *KeyAny* and *ValueAny*.

KeyAny Any. Key object to associate with *ValueAny* within dictionary *AnyDict*.

ValueAny Any. Value object to associate with *KeyAny* within dictionary *AnyDict*.

AnyStr String. String object into which to store *CharInt*.

CharInt Integer. Integer value of character to store within string *AnyStr* at index *IndexInt*.

Comments

In the first variant, the operator store the object *ElementAny* at index *IndexInt* within the array *AnyArray*. Array indexes range from zero to N-1, where N is the number of elements in the array.

In the second variant, the operator stores the key object *KeyAny* and the value object *ValueAny* as a key + value pair into the dictionary *AnyDict*. If *AnyDict* already contains a key + value pair with a matching key object, put replaces the entry's associated value object with *ValueAny*.

In the third variant, the operator stores the ASCII character with the value *CharInt* within the string *AnyStr* at the index *IndexInt*. String character indexes range from zero to N-1, where N is the number of characters in the string.

putinterval

Description

Store a range of data into an array or string object.

Usage

AnyArray IndexInt SubArray **putinterval**
AnyStr IndexInt SubStr **putinterval**

AnyArray Array. Array object to receive sub-array objects.

AnyStr String. String object to receive sub-string characters.

IndexInt Integer. When used with *AnyArray*, index of first data object to replace within array. Arrays begin with index zero. When used with *AnyStr*, index of first character to replace within string. Strings begin with index zero.

SubArray Array. Array containing objects to replace sub-range within *AnyArray*.

SubStr Any. String containing characters to replace sub-range within *AnyStr*.

Comments

In the first variant, the operator replaces objects starting at index *IndexInt* within the array *AnyArray*. The replacement continues for the number of elements in *SubArray*. Array indexes range from zero to N-1, where N is the number of elements in the array. The operator replaces the *AnyArray* objects with duplicates of objects in the array *SubArray*. For composite objects, the duplicate objects share their data with the original objects in the *SubArray*.

In the second variant, the operator replaces characters starting at index *IndexInt* within the string *AnyStr*. The replacement continues for the number of characters in *SubStr*. String character indexes range from zero to N-1, where N is the number of characters in the string. The operator replaces the *AnyStr* characters with the characters from *SubStr*.

quit

Description

Terminate the PAL interpreter.

Usage

```
quit
```

Comments

This operator generally serves no purpose when used with a PAL printer. Different PAL printers respond to this operator in different ways. Some printers simply restart the PAL interpreter after discarding all PAL data stored within the printers memory. Other printers will simply halt execution. In general, programmers should not use this operator.

readstring

Description

Read data from an open file.

Usage

OpenFile AnyStr **`readstring`** *ReadStr GoodBool*

OpenFile	File. A file object returned by the file operator. The programmer must have opened the file for reading.
AnyStr	String. A string which establishes the number of bytes to read from the specified file. The data contained within the string does not matter. PAL only uses the length of the string to determine the number of bytes to read.
ReadStr	String. Data read from file. The length of the string will depend upon the number of bytes available from the file. The string will not exceed the length of *AnyStr*.
GoodBool	Boolean. true if read was successful. false if no data available due to end of file.

Comments

PAL only uses the length of *AnyStr* to determine the number of bytes to read from the specified file. PAL ignores any data contained in *AnyStr*.

PAL will return a string which does not exceed the length of *AnyStr*. If the file does not have sufficient data immediately available, readstring will only return the immediately available data. readstring does not block waiting for the requested amount of data to become available.

PAL generally considers as immediately available all data contained in files on some form of storage device. Typically only files associated with some form of input device will not have data available immediately.

If PAL returns *GoodBool* as false, it indicates that PAL reached the end of the file without recovering any data. PAL will return true even if PAL encounters the end of file after reading part of the requested data. The file must have no data remaining for PAL to return in order for PAL to return *GoodBool* as false.

PAL will typically not return an *GoodBool* value of false for file associated with most input devices. For most input devices, PAL assumes that new data may arrive via the device at a future time.

repeat

Description

Executes the specified procedure for the specified number of iterations.

Usage

CountInt AnyProc **repeat**

CountInt Integer. Specifies the number of times the interpreter should execute *AnyProc*.

AnyProc Procedure. The procedure which the interpreter will repetitively execute.

Comments

PAL first removes the supplied iteration count and procedure from the stack. The interpreter then executes the procedure for the specified number of iterations. The repeat operator does not place any objects on the operand stack. However, the procedure may place objects on the stack if desired.

The programmer can use the exit operator to prematurely terminate a repeat loop.

Hints

The following three PAL sequences perform entirely different functions.

```
1: 5 MyProc repeat
2: 5 /MyProc repeat
3: 5 {MyProc} repeat
```

The first example instructs PAL to execute the procedure MyProc *before* PAL executes the operator repeat. As a result, the repeat operator will generate an error unless MyProc places a procedure object onto the stack before terminating.

In the second example, when the repeat operator executes, it will encounter a literal name (/MyProc) on the stack. Since repeat expects a procedure object, this will produce an error.

The third example shows the proper approach to repetitively execute the procedure MyProc. The repeat operator will execute the procedure "{MyProc}" five times. The procedure, in turn, executes the procedure MyProc during each iteration.

The specified procedure may also do more than just execute a saved procedure. The following example sums ten numbers on the operand stack without removing the values from the stack.

```
10 {9 index} repeat 9 {add} repeat
```

rlineto

Description

Appends a line to the current path which extends from the current point to a point a given distance away from the current point.

Usage

XDeltaNum YDeltaNum **`rlineto`**

XDeltaNum Integer or fixed-point. Specifies the X distance over which to extend the line.

YDeltaNum Integer or fixed-point. Specifies the Y distance over which to extend the line.

Comments

The rlineto operator does not actually draw a line on the current page. Instead, it adds the line to a drawing *path* which it keeps in memory. As the programmer specifies additional drawing operations, PAL continues to append these operations to the end of the drawing path. Once the programmer has specified the entire path for PAL to draw, the programmer uses the stroke operator to instruct PAL to actually *stroke* the path onto the page. PAL also provides the setlinewidth and setlinecap operators which influence the stroke operator.

In general, the programmer should not use rlineto to finish a path around a closed object such as a square. For a square, the drawing sequence should use rlineto and/or lineto to draw the first three sides of the square, but closepath for the final side. closepath instructs PAL to join the end of the closepath line to the first point of the path. This allows PAL to smooth the transition from the last line to the first line during the stroke operation.

After appending the rlineto operation to the path, PAL updates the current point in the graphics state to the X and Y coordinate at the specified deltas from the initial coordinate. This allows subsequent drawing orders to automatically continue from the end of the line.

PAL currently restricts line drawing to horizontal and vertical lines. Therefore, in order to receive expected results, the programmer should always specify zero for either the *XDeltaNum* or *YDeltaNum* parameter.

rmoveto

Description

Establishes the new current point at a relative distance from the current point.

Usage

XDeltaNum YDeltaNum **`rmoveto`**

XDeltaNum Integer or fixed-point. Relative distance along the X axis to the new current co-ordinate. Specified in user coordinates.

YDeltaNum Integer or fixed-point. Relative distance along the Y axis to the new current co-ordinate. Specified in user coordinates.

Comments

Most PAL drawing operators use the current point as the location at which to draw. In most cases, the current point establishes the bottom left corner of any new object drawn.

The rmoveto operator also has the affect of ending any active sub-path in the graphics state and starting a new sub-path. Depending upon the desired result, the programmer may wish to specify closepath before a rmoveto in order to close the active sub-path. Using rmoveto without closepath creates an open sub-path.

Closing a sub-path instructs PAL to smooth the connection between the first and last lines of the path. PAL assumes that the start and end points of an open path either do not meet or the programmer does not wish PAL to smooth the connection.

rotate

Description

Rotates the user coordinate system about the user coordinate system origin.

Usage

AngleNum **`rotate`**

AngleNum Integer or fixed-point. Number of degrees to rotate the user coordinate system. A positive angle indicates a counter-clockwise rotation. A negative angle indicates a clockwise rotation. The programmer should restrict the value to 90 degree increments in order to ensure proper operation.

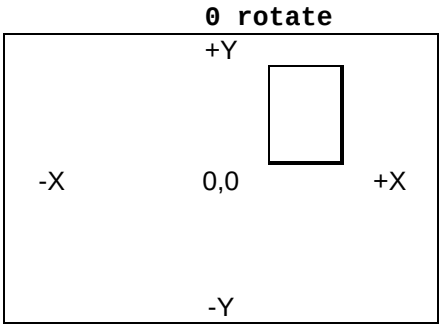
Comments

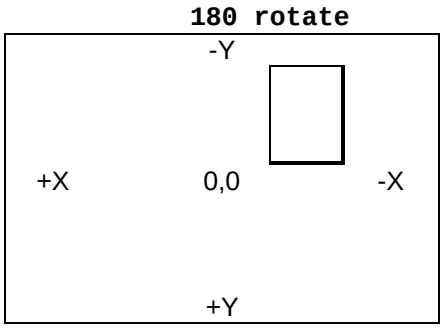
PAL applies the rotation to the current user coordinate system. Therefore, rotations accumulate. The orders "90 rotate 90 rotate" perform the same function as "180 rotate".

Two approaches exist to using the rotate operator. The first approach works well when designing a page which contains only a few images rotated differently from the majority of the images. The second approach works well when drawing a large number of images in a rotated orientation.

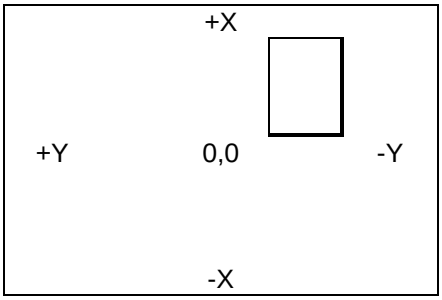
The first approach takes advantage of the fact that the rotate operator only affects the operation of future operators. It does not affect the current point or the points in the current path. Therefore, the programmer can use the moveto operator to establish the current point using the programmer's preferred rotation for the majority of the images. After establishing the current point, the programmer can issue the rotate operator to draw the next image in the alternate rotation. After PAL has drawn the rotated image, the programmer can issue the negative of the prior rotation to restore the original drawing and positioning orientation.

The second approach allows the programmer to actually design the page in a rotated orientation. When using this approach, the programmer will probably wish to combine a translate operation with any rotate. The following diagrams illustrate the effect of rotating the user coordinate system without performing an accompanying translate. The small rectangles show the position of the page in relation to the user coordinate system

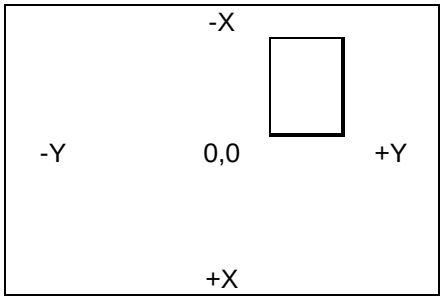




90 rotate



270 rotate

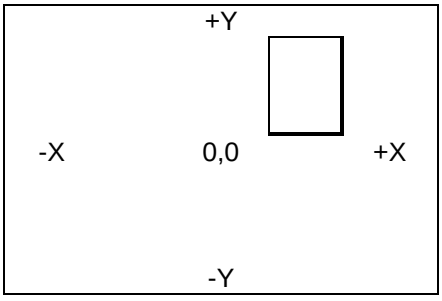


As the reader can see from the diagrams, with the origin at its default position, only zero rotation places the +X, +Y quadrant over the page. Although PAL fully supports drawing using negative coordinates, most programmers find it easier to use only positive coordinates. As a result, most programmers will prefer to relocate the coordinate system origin to position the +X, +Y quadrant over the page.

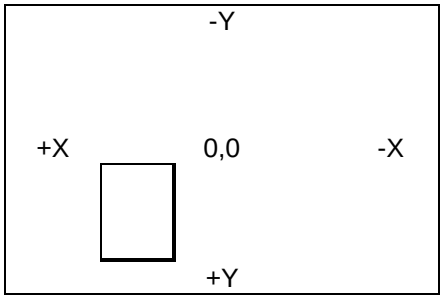
The following table shows the appropriate rotate and translate operations to adjust the coordinate system for drawing in all four orientations. The *W* and *H* values specify the width and height, respectively, of the page in user coordinates. The starting orientations assume that the entries in this table were used to reach that starting orientation.

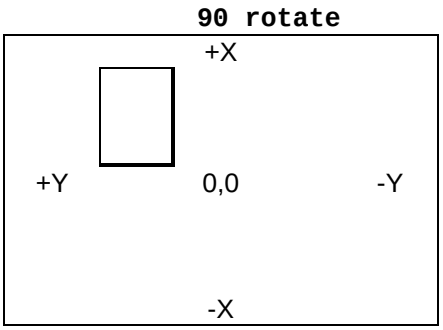
Starting Orientation	Desired Orientation	Rotation (first)	Translation (second)
0	90	90 rotate	0 *-W* translate
0	180	180 rotate	*-W* *-H* translate
0	270	270 rotate	*-H* 0 translate
90	0	-90 rotate	*-W* 0 translate
90	180	90 rotate	0 *-H* translate
90	270	180 rotate	*-H* *-W* translate
180	0	-180 rotate	*-W* *-H* translate
180	90	-90 rotate	*-H* 0 translate
180	270	90 rotate	0 *-W* translate
270	0	-270 rotate	0 *-H* translate
270	90	-180 rotate	*-H* *-W* translate
270	180	-90 rotate	*-W* 0 translate

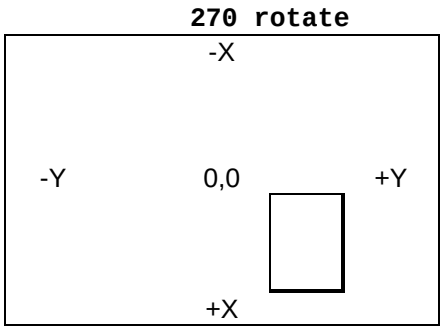
0 rotate



180 rotate







Hints

The following two examples produce the exact same label image. They both draw the word "Sideways" starting at 72,72 and running from the bottom of the page toward the top.

```
72 72 moveto
90 rotate
(Sideways) show
-90 rotate

90 rotate
72 -72 moveto
(Sideways) show
-90 rotate
```

The relationship between the first rotate and the moveto demonstrates the two approaches to using the rotate operator. The first example demonstrates the first approach. The moveto operator first establishes the current point using the existing coordinate system. The rotate operator then rotates the coordinate system, but does not affect the current point.

The second example demonstrates the second approach. The example starts by rotating the coordinate system. As a result, the programmer must specify the coordinates for the moveto operator in the new rotated coordinate system. Examining the coordinate system diagrams provided above in conjunction with the example will show why the Y coordinate requires a negative value.

round

Description

Rounds the specified value to the nearest integer.

Usage

AnyNum **round** *RoundedNum*

AnyNum	Integer or fixed-point. Value to round to the nearest integer.
RoundedNum	Integer or fixed-point. Nearest integer to *AnyNum*. The type of the returned value matches the type of the supplied parameter.

Comments

Although this operator will accept integer values, this operator has no affect upon integers. The following table shows the affect of the round operator upon various fixed-point values.

```
 1.6  round  2.0
 1.5  round  2.0
 1.4  round  1.0
 1.0  round  1.0
 0.0  round  0.0
-1.0  round -1.0
-1.4  round -1.0
-1.5  round -1.0
-1.6  round -2.0
```

_rtrim

Description

Eliminate undesired characters from right (trailing) edge of a string.

Usage

AnyStr SetStr **_rtrim** *TrimmedStr*

AnyStr	String. String possibly containing character to eliminate.
SetStr	String. Set of undesired characters. An empty string "()" instructs the interpreter to trim whitespace characters.

Comments

PAL creates the new string *TrimmedStr* by copying the contents of *AnyStr*. As PAL copies the characters from right to left, PAL compares each character to the list of characters contained in *SetStr*. If PAL finds the character in *SetStr* it does not copy the character to *TrimmedStr*. As soon as PAL finds a character from *AnyStr* which does not match a character in *SetStr*, PAL then copies that character, and all remaining *AnyStr* characters to *TrimmedStr*.

Specifying an empty string "()" for *SetStr* instructs PAL to trim all whitespace characters from the right edge of *AnyStr*. PAL treats all characters with a decimal value of 32 and below as whitespace characters. This includes ASCII spaces, tabs, carriage returns, line feeds, as well as all other standard ASCII non-printable control characters.

scale

Description

Independently alter the scale factors for the user coordinate system along the X and Y axis.

Usage

XScaleNum YScaleNum **`scale`**

XScaleNum Integer or fixed-point. Factor by which to adjust scaling along the X axis.

YScaleNum Integer or fixed-point. Factor by which to adjust scaling along the Y axis.

Comments

PAL applies the new scale factors to the current user coordinate system. Therefore, scale factors accumulate. The orders "2 4 scale 4 8 scale" produce the same affect as "8 32 scale".

Hints

The scale operator allows the programmer to customize the coordinate system. For example, United States programmers might prefer to use inches as their standard unit of measure. A U.S. programmer can quickly adjust the user coordinate scale factor from points (1/72") to inches using the order "72 72 scale".

On the other hand, programmers in other parts of the world might prefer to use millimeters as their standard unit of measure. These programmers can quickly adjust the user coordinate scale factor from points to millimeters using the order "72 25.4 div 72 25.4 div scale".

The programmer should take note that changing the coordinate system scale factor also affects fonts. Initially, a font dictionary indicates that PAL should draw the font with a height of one user unit. Under the default coordinate system, this means one point (1/72").

If the user changes the coordinate system to, for example, one inch, the font dictionary will then specify a one inch tall font. Under the default coordinate system, the programmer would use the operation "12 scalefont" to make the font 12 points tall. However, if the programmer did this with an one inch coordinate system, it would specify a 12 inch tall font. Instead, the programmer would need to use "12 72 div scalefont" to select a 12 point font.

scalefont

Description

Modifies a font dictionary to scale the characters to a desired height.

Usage

AnyFontDict ScaleNum **`scalefont`** *ScaledFontDict*

AnyFontDict Dictionary. Font dictionary returned by the findfont operator.

ScaleNum Integer or fixed-point. Factor by which to scale characters of the font.

ScaledFontDict Dictionary. *AnyFontDict* modified to scale the font's characters to the desired height. PAL modifies *AnyFontDict*. It does not create a new dictionary.

Comments

The font dictionary returned by the findfont operator contains information designed to scale the characters of the font to one user coordinate tall. The scalefont operator modifies the font dictionary to draw characters of any desired height.

The font dictionary contains only relative character scaling information. PAL combines the font's scaling information with the current user scaling factor to determine the true size of the characters. PAL combines this information when it draws the characters.

Since fonts default to one user coordinate tall, and the user coordinate system defaults to one point scaling, each character defaults to a height of one point. However, changing the user coordinate scaling factor will accordingly change the default character height. For example, changing the user coordinate system to inches changes the default character height to one inch.

The scalefont operator allows the programmer to compensate for the current user coordinate system scaling as well as establish any desired height for the characters. Under the default coordinate scaling of points, the "*AnyFontDict* 12 scalefont" operation will configure the font information to draw 12 point characters. However, the same scalefont operation with a user coordinate system based on inches would produce 12 inch tall characters. Therefore, with a user coordinate system based on inches, "*AnyFontDict* 12 72 div scalefont" will configure the font information to draw 12 point characters.

Hints

PAL does not automatically scale all the characters of a font in response to the scalefont operator. Since character scaling takes time, PAL waits to scale each character until it needs to draw the character onto a page. PAL also rotates the characters at this time to match the orientation at which it must draw the characters.

Once PAL scales and rotates a character, it places the character image into a *character cache*. This allows PAL to use the already scaled character image if PAL must draw the character again.

PAL keeps the scaled and rotated character images in a special private area of *ScaledFontDict*. So long as a reference to *ScaledFontDict* exists within the printer's memory, PAL will retain the characters it has already scaled and rotated.

The setfont operator creates a reference within the graphics state to the specified *ScaledFontDict*. Therefore, so long as the graphics state continues to reference a given *ScaledFontDict*, PAL will retain the character images associated with the font.

However, selecting a new font using the setfont operator causes PAL to replace the graphics state's current font dictionary reference with a reference to the new font dictionary. As a result, the graphics state no longer references the previous font dictionary. Unless the PAL programmer has created another reference to the previous font's *ScaledFontDict*, PAL will automatically discard the old font dictionary as well as any scaled character images associated with it.

For simple printing applications which require only a single font, managing the character cache presents little problem. The graphics state's reference to the single *ScaledFontDict* maintains all scaled character images.

For more complex printing applications which use multiple fonts, the PAL programmer should save each *ScaledFontDict* for as long as the programmer requires the font to ensure that PAL retains all character images. The following example shows a simple example of how to manage three fonts under PAL.

```
1: /Font1 /PALFont1 findfont 12.00 scalefont def
2: /Font2 /PALFont2 findfont 10.00 scalefont def
3: /Font3 /PALFont3 findfont 18.00 scalefont def
4: 72 144 moveto Font1 setfont (Font1 text) show
5: 72 134 moveto Font2 setfont (Font2 text) show
6: 72 116 moveto Font3 setfont (Font3 text) show
```

The names *PALFont1*, *PALFont2*, *PALFont3* represent any given font available on a given PAL printer. Lines one through three locate and establish scaling factors for three arbitrary fonts. Each line then saves the *ScaledFontDict* associated with each font under the names Font1, Font2, and Font3.

Lines four through six use each of these three fonts in turn. When line five performs the sequence "Font2 setfont", it instructs the interpreter to replace the graphics state reference to Font1 with a reference to Font2.

At this point, if line one had not saved Font1's *ScaledFontDict*, PAL would have normally discarded the *ScaledFontDict* associated with Font1. This would also result in PAL discarding any cached characters associated with *ScaledFontDict*. However, since line one does save *ScaledFontDict*, the reference to the *ScaledFontDict* created causes the interpreter to maintain the *ScaledFontDict* in memory. As a result, PAL also maintains the scaled and rotated characters associated with *ScaledFontDict*.

PAL maintains only one image for each character associated with a given *ScaledFontDict*. Therefore, the programmer should save separate *ScaledFontDict* dictionaries for every combination of font, point size, and rotation used.

search

Description

Searches for the first occurance of one string within another string.

Usage

AnyStr SearchStr **search** *PostStr MatchStr PreStr* **true**
AnyStr SearchStr **search** *AnyStr* **false**

AnyStr String. Search to search.

SearchStr String. String to locate within *AnyStr*.

PostStr String. Characters from *AnyStr* which follow *SearchStr*.

PreStr String. Character from *AnyStr* which precede *SearchStr*.

MatchStr String. Characters from *AnyStr* which match the characters from *SearchStr*. *MatchStr* will contain the same characters as *SearchStr*, but *MatchStr* will exist as an entirely different string.

true Boolean. Indicates that the interpreter found *SearchStr* within *AnyStr*.

false Boolean. Indicates that the interpreter was unable to locate *SearchStr* within *AnyStr*.

Comments

The interpreter will stop when it locates the first occurance of *SearchStr* within *AnyStr*. If the interpreter locates *SearchStr* within *AnyStr*, the interpreter returns three new strings followed by true. true informs the PAL procedure that a match was found. The three new strings contain the characters before the matching characters, the matching characters themselves, and the characters which follow the matching characters.

Since *MatchStr* contains the characters which matched *SearchStr*, *MatchStr* will contain the same characters as *SearchStr*. However, *MatchStr* will exist in memory as an entirely different string from *SearchStr*. Any subsequent modifications to either string will not affect the other string.

If the interpreter cannot locate *SearchStr* within *AnyStr*, the interpreter returns *AnyStr* followed by false. false informs the PAL procedure that a match was not found.

setfileposition

Description

Relocates a file's read/write pointer.

Usage

OpenFile PositionInt **setfileposition**

OpenFile File. File object for open file to relocate read/write pointer.

PositionInt Integer. New offset for file's read/write pointer from start of file. A value of zero indicates the first byte of the file.

Comments

This operator allows the programmer to randomly read and write various locations of a direct access file. This operator has no affect on sequential files. Sequential files include all of the standard PAL files.

Hints

The following example uses setfileposition to write the string "olleH" to the file MyFile.

```
MyFile 4 setfileposition MyFile (H) writestring
MyFile 3 setfileposition MyFile (e) writestring
MyFile 2 setfileposition MyFile (l) writestring
MyFile 1 setfileposition MyFile (l) writestring
MyFile 0 setfileposition MyFile (0) writestring
```

setfont

Description

Establishes the specified font as the current font to use for drawing characters.

Usage

ScaledFontDict **`setfont`**

ScaledFontDict Dictionary. A font dictionary previously scaled by the scalefont operator.

Comments

In order to draw characters of a given font, the programmer must first use the findfont operator to recover the desired font. Next, the programmer uses the scalefont operator to establish the desired height for the characters. Finally, the programmer must establish the font as the font to use for all future character drawing operations. The setfont operator performs this final operation.

setgray

Description

Establishes a gray scale printing level within the DeviceGray color space to use for subsequent image rendering operations.

Usage

LevelNum **`setgray`**

LevelNum Integer or fixed-point. Specifies the gray level (amount of white) to apply to subsequent image rendering operations. *LevelNum* can range from 0.0 (black) to 1.0 (white).

Comments

PAL currently only supports the DeviceGray color space. This color space allows the user to select a gray level which PAL will apply to all subsequent drawing operations. The gray level setting specifies the percentage of white to apply to images. A setgray values of 0.0 specifies 0% white, which implies black. A setgray value of 1.0 specifies 100% white. Intermediately values select intermediate levels of gray.

No printer can support the full range of gray levels which the user can specify using the setgray operator. As a result, all PAL printers translate ("map") gray level requests to the gray levels the printer can print. This translation process is referred to as "color realization." Color realization involves the conversion of a color from a "virtual" or "logical" requested color to a "real" or "physical" color which the printer can actually print.

Different printers will perform color realization in different manners depending upon their printing capabilities. Some printers will only support black and white printing. These printers will typically translate gray level requests below 0.5 (50%) to black, and requests at or above 0.5 to white.

Some printers will create a pattern of black and white dots. The relative proportion of black dots to white dots can produce the visual effect of varying levels of gray. The most advanced printers may actually have the ability to vary the level of black applied to each dot on the page.

In general, the user can rely upon PAL printers supporting both black and white printing as a minimum. Specifying "0 setgray" will always select black, and "1 setgray" will always select white.

setlinecap

Description

Controls the drawing of line end caps during a stroke operation.

Usage

CapStyleInt **`setlinecap`**

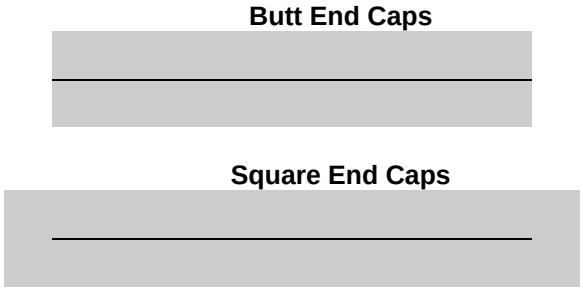
CapStyleInt Integer. Selects the desired line cap style. A value of 0 selects butt end caps. A value of 2 selects square end caps.

Comments

The graphics state contains various parameters which influence the manner in which the stroke operator renders a path onto the page. setlinecap allows the programmer to change the line end cap style specified in the graphics state.

The stroke operator uses the current line end cap style when drawing the starting and ending points of open paths. For a closed path, PAL joins together the starting and ending points, so no end points exist.

PAL currently provides two styles of line end caps — butt and square. The illustrations provided below show the difference between the two styles. The gray area shows the actual line drawn by the stroke operator based on the current line width. The solid line shows the center line as specified using the lineto operator.



The current line cap style has no affect until PAL encounters the stroke operator. Therefore, the programmer may specify setlinecap either before or after using the various drawing operators to build the desired path.

PAL uses the butt end cap style by default.

setlinewidth

Description

Controls the width, or thickness, of lines drawn using the stroke operator.

Usage

WidthNum **`setlinewidth`**

WidthNum Integer or fixed-point. Specifies the desired line width, or thickness, in user coordinate system units.

Comments

The graphics state contains various parameters which influence the manner in which the stroke operator renders a path onto the page. setlinewidth allows the programmer to change the line width, or thickness, specified in the graphics state.

During a stroke operation, PAL uses the line information contained in the current path to establish the center of lines on the page. Therefore, as shown below, the line will extend for ½ *WidthNum* to both sides of the center line.

PAL uses a default line width of one user coordinate system unit.

Hints

If the programmer scales the user coordinate system, the programmer will probably also wish to change the line width before drawing any lines. For example, if the user changes the user coordinate system to inches, the programmer probably will not want to draw one inch wide lines.

No matter how small a line width the programmer specifies, PAL guarantees to draw lines at least one pixel thick. Specifying a line width of zero instructs PAL to draw hairlines. PAL draws hairlines as a single pixel thick regardless of the resolution of the device. This may not work well on extremely high resolution devices such as photo-typesetters. However, it will work fine on common computer printers.

_setlocaltime

Description

Allows the user to set the printer's internal real-time clock via PAL commands.

Usage

TimeArray **_setlocaltime**

TimeArray Array. Contains the new time information in a format compatible with the time information returned by the _localtime operator.

[*AvailBool TotalInt YearInt MonthInt DayInt HourInt MinInt SecInt Sec100Int DOYInt DOWInt ZoneNum SaveInt***]**

AvailBool Boolean. **true** if *TimeArray* actually contains values available for recording into the printer's real-time clock electronics. **false** if the printer should not record the values in *TimeArray* to the clock electronics.

TotalInt Integer. Any non-negative value. The _setlocaltime operator only requires a non-negative value for this value. The operator does not use this value when setting the clock electronics. Required only for compatibility with _localtime.

YearInt Integer. 1970..2970. The _setlocaltime operator uses this value to set the current year within the printer's clock electronics.

MonthInt Integer. 1..12. The _setlocaltime operator uses this value to set the current month or the year within the printer's clock electronics.

DayInt Integer. 1..31. The _setlocaltime operator uses this value to set the current day of the month within the printer's clock electronics.

HourInt Integer. 0..23. The _setlocaltime operator uses this value to set the current hour of the day within the printer's clock electronics.

MinInt Integer. 0..59. The _setlocaltime operator uses this value to set the current minute of the hour within the printer's clock electronics.

SecInt Integer. 0..59. The _setlocaltime operator uses this value to set the current second of the minute within the printer's clock electronics.

Sec100Int Integer. 0..99. The _setlocaltime operator uses this value to set the current hundredth of the second within the printer's clock electronics. Not all PAL printer models support time resolution to within 1/100 of a second. The user should consult documentation specific to each printer model to determine the various resolutions supported.

DOYInt Integer. 1..366 or 0. Not used to set clock electronics. Required for compatibility with _localtime.

DOWInt	Integer. 1..7 or 0. Not used to set clock electronics. Required for compatibility with _localtime.
ZoneNum	Integer or fixed-point. 0..23 or -1. If a fixed-point value is used, the fractional portion of the value must be zero at this time.
SaveInt	Integer. 0, 1. The use of this value is subtly, but *very* significantly different from its _localtime counterpart. See comments below.

Comments

Setting *AvailBool* to **`false`** effectively causes the _setlocaltime operator to not perform any operation. However, the operator will still require that *TimeArray* contain the appropriate number of elements. This interpretation of the *AvailBool* entry in *TimeArray* provides the proper complement to the *AvailBool* entry returned by the _localtime operator.

_setlocaltime does not use *DOYInt* to set the current day of the year. For the _localtime operator, the printer always calculates the day of the year from the other time values. The _setlocaltime operator does impose the 1..366 range, or a filler value of zero, restriction upon *DOYInt* value. The operator requires this *TimeArray* entry strictly for compatibility with the *TimeArray* operand returned by the _localtime operator.

_setlocaltime does not use *DOWInt* to set the current day of the week. For the _localtime operator, the printer always calculates the day of the week from the other time values. The _setlocaltime operator does impose the 1..7 range, or a filler value of zero, restriction upon this value. The operator requires this *TimeArray* entry strictly for compatibility with the *TimeArray* operand returned by the _localtime operator.

ZoneNum establishes the relationship between the local time and Greenwich mean time (GMT). The user can use a value of -1 to indicate that the relationship is either not important to the user's use of the printer, or is not known and, by implication, not important to the user's use of the printer. Establishing this value allows printers and/or systems in different time zones to resolve their time relationships.

The _setlocaltime and _localtime operators interpreter the meaning of the *SaveInt* parameter in different ways. However, the difference is very subtle and the user should play close attention to this dicussion.

The *SaveInt* value return by _localtime indicates whether or not the current time is for daylight savings time or standard time. The _setlocaltime *SaveInt* value is used to indicate whether the locality uses daylight savings time during the year. Therefore, if the locality uses daylight savings time but it is currently standard time at that locality, _localtime will return **`false`** for *SaveInt* since the current time is not daylight savings time. However, the user would still specify **`true`** for the _setlocaltime *SaveInt* value since the locality uses daylight savings time.

The real-time clock electronics in many PAL printers support the automatic updating of their internal time during the cross overs between daylight savings time and standard time. The _setlocaltime *SaveInt* value indicates whether or not the printer should enable this feature within the printer's electronics. Also, the _localtime operator will only return a **`true`** value during daylight savings time if the user has instructed the printer that the locality uses daylight savings time.

For the _setlocaltime operator, a *SaveInt* value of zero instructs the printer to disable support for daylight savings time. A value of one instructs the printer to enable support for daylight savings time.

setpagedevice

Description

Provides very low-level control over page printing mechanism.

Usage

ControlDict **setpagedevice**

ControlDict Dictionary. Specifies various new settings to control page printing.

/CutMedia Integer. Only supported on continuous media printers which have media cutting capability. A value of zero (0) instructs the printer not to cut the media. A value of four (4) instructs the printer to cut the media between every page.

/_FeedRate Integer or fixed-point. Controls the speed at which the printing mechanism feeds the media. The entry specifies the feed rate as a percentage of the maximum feed rate supported by the mechanism. A value of 100 implies full speed printing. A value of 0 implies no feeding of the media. Typically, printers which support this parameter also enforce a minimum feed rate. As a result, percentages which result in a feed rate below this minimum will only result in the minimum feed rate.

/_HorzAlign Integer or fixed point. *This feature is only available on the Fastmark series of printers.* Allows the printing to be shifted in the horizontal direction. This parameter is specified in inches and effectively sets the Horz Align feature available through the front panel.

/ImagingBBox Array. Establishes the dimensions of the logical printable area and the logical printable area's relationship to the physical media. PAL expects all values in points (1/72"). The user coordinate system has no affect on this parameter. If the logical printable area exceeds the physical printable area for the active media, PAL will reduce the logical printable area as necessary. *On the Fastmark printer line, this parameter has no effect.*

[*HorzOrg VertOrg Width Height***]**

HorzOrg Integer or fixed-point. Establishes the location of the logical printable area's left edge in relation to the physical media's left edge. The user must specify this value in points (1/72").

VertOrg Integer or fixed-point. Establishes the location of the logical printable area's bottom edge in relation to the physical media's bottom edge. The user must specify this value in points (1/72").

Width Integer or fixed-point. Establishes the width of the logical printable area in points (1/72").

Height Integer or fixed-point. Establishes the height of the logical printable area in points (1/72").

/_ImprintLevel Integer or fixed-point. In general, this parameter specifies the amount of energy used to apply the print image to the page. The entry specifies the imprinting level as a percentage of the maximum imprinting level supported by the mechanism. A value of 100 implies the maximum imprinting level supported. A value of 0 implies no imprinting. Typically, printers which support this parameter also enforce a minimum imprinting level. As a result, percentages which result in a imprinting level below this minimum will only result in the minimum imprinting level. Only certain PAL printers support this parameter. The exact meaning of the parameter varies depending upon the printing mechanism.

/_InterPage Integer or fixed-point. Specifies the distance, in points (1/72"), between pages on continuous form media. The current user scaling factor has no affect upon this value. Only certain continuous forms printers support this parameter.

/ManualFeed Boolean. Selects between manual feed and automatic feed operation. true selects manual feed operation. false selects automatic feed operation. The exact meaning of this parameter varies between printers. On cut sheet printers, true usually indicates that the printer should wait for the operator to supply each sheet. On continuous media printers, true usually indicates that the printer should wait for the operator to remove the previous page printed before printing the next page. This parameter is used to enable/diable the present sensor for peel and present printers. Not all printers support this parameter.

/_ManualSenseLevel
Integer or fixed-point. Controls the sensitivity of the sensor used on some printers to detect completion of the manual feed operation by the printer operator. The value specifies a percentage of the maximum sensitivity provided by the sensor. A value of 100 implies absolute sensitivity. A value of 0 implies no sensitivity. Absolute sensitivity will normally result in the printer constantly believing that the operator has completed the manual feed operation. No sensitivity will normally prevent the printer from ever detecting that the user has completed the manual feed operation. The proper setting will normally fall somewhere closer to the middle of the range, depending upon the operating conditions. Not all printers support this parameter. *This parameter is ignored on the Fastmark series of printers.*

/_MediaSenseType
String. Selects the media page sensing mechanism. Possible values are (BlackBar), (Gap) or (Continuous).

/_MediaOutSenseLevel
Integer or fixed-point. Controls the sensitivity of the sensor used on some printers to detect the presence of media within the printer. The value specifies a percentage of the maximum sensitivity provided by the sensor. A value of 100 implies absolute sensitivity. A value of 0 implies no sensitivity. Absolute sensitivity will normally result in the printer constantly believing that the printer contains media ready for printing.

No sensitivity will normally prevent the printer from ever detecting the presence of media within the machine. The proper setting will normally fall somewhere closer to the middle of the range, depending upon the operating conditions. Not all printers support this parameter. *This parameter is ignored on the Fastmark series of printers.*

/MediaType String. Selects or specifies the desired input media. On printers capable of sensing the input media, this parameter operates in a selection mode. It allows the programmer to request a particular type of media. For example, it allows selection between multiple input bins on a multi-bin sheet-fed printer. On printers not capable of sensing the input media, this parameter operates in a specification mode. It allows the programmer to instruct the printer to assume the presence of a particular type of media. Programmers should check documentation specific to each printer to determine the operation of this parameter as well as the valid string values. On the Fastmark printer line, the possible values for this parameter are *(Direct*) for direct thermal media and *(Transfer*) for thermal transfer media.

/OutputType String. Selects or specifies the desired media output destination. This typically implies which of multiple output bins should receive the printed pages. Programmers should check documentation specific to each printer to determine the operation of this parameter as well as the valid string values. *This parameter is ignored on the Fastmark series of printers.*

/_PageSenseLevel

Integer or fixed-point. Controls the sensitivity of the sensor used on some printers to detect the start and end of a page. The value specifies a percentage of the maximum sensitivity provided by the sensor. A value of 100 implies absolute sensitivity. A value of 0 implies no sensitivity. Absolute sensitivity will normally result in the printer detecting the start and end of pages at the wrong times. No sensitivity will normally prevent the printer from ever detecting the start and end of pages. The proper setting will normally fall somewhere closer to the middle of the range, depending upon the operating conditions. Not all printers support this parameter. *This parameter is ignored on the Fastmark series of printers.*

/PageSize Array. Establishes the dimensions of the media in points (1/72"). The user coordinate system has no affect on this parameter. Printers which can detect media sizes may use this parameter to select the closest installed media. Printers which cannot detect media sizes may rely upon this parameter as the actual size of the media.

[*Width Height***]**

Width Integer or fixed-point. Establishes the width of the media in points (1/72").

Height Integer or fixed-point. Establishes the height of the media in points (1/72").

/_PigmentSenseLevel

Integer or fixed-point. Controls the sensitivity of the sensor used on some printers to detect the presence of the printing pigment. The actual pigment used varies by printer technology. Pigment can include inked ribbons, toner, thermal transfer ribbons, or other technologies. The value specifies a percentage of the maximum sensitivity provided by the sensor. A value of 100 implies absolute sensitivity. A value of 0 implies no sensitivity. Absolute sensitivity will normally result in the printer constantly believing that the printer contains suffient pigment to print a page. No sensitivity will normally prevent the printer from detecting the presence of the pigment. The proper setting will normally fall somewhere closer to the middle of the range, depending upon the operating conditions. Not all printers support this parameter. *This parameter is ignored on the Fastmark series of printers.*

/_PresentDistance

Integer or fixed point. *This feature is only available on the Fastmark series of printers.* Allows the presentation distance to be set.

/_VertAlign Integer or fixed point. *This feature is only available on the Fastmark series of printers.* Allows the printing to be shifted in the vertical direction. This parameter is specified in inches and effectively sets the Vert Align feature available through the front panel.

Comments

This operator performs extremely printer dependent configurations. The programmer should consult each printer's documentation to determine the relationship of these parameters to the operation of the printer.

PAL always performs initgraphics and erasepage operations while processing the setpagedevice operator.

show

Description

Draw text on the page at the current coordinate using the current font.

Usage

ShowStr **show**

ShowStr String. Text for PAL to draw at the current coordinate.

Comments

The programmer can use the moveto or rmoveto operator to place the current coordinate at the desired position for drawing the text. PAL provides the findfont, scalefont, and setfont operators for establishing the desired font to use for drawing text.

The current coordinate establishes the *inline* and *baseline* position of the first character drawn. For Roman characters, the inline position specifies the left edge of the character. The baseline position establishes the imaginary line upon which PAL will draw the characters. Characters with decenders will drop below the baseline.

For non-Roman fonts, PAL will treat the inline and baseline positions in a manner appropriate to the font.

PAL automatically updates the current coordinate to the appropriate new inline and baseline positions following the last character drawn.

showpage

Description

Print physical page.

Usage

```
showpage
```

Comments

PAL does not actually print any images onto the physical page until it receives the showpage operator. The programmer sends the showpage operator to inform PAL that the interpreter has received all drawing orders for the current page. Since no more drawing will occur on the current page, PAL may now print the various images previously received.

By default, the showpage operator prints only a single page. The programmer may use the _showpages operator to instruct PAL to print multiple copies of a single page image.

Once PAL has printed the page, it automatically performs erasepage and initgraphics operations.

_showpages

Description

Print multiple copies of a single page image.

Usage

PagesNum **_showpages**

PagesNum Integer or fixed-point. Specifies the number of identical copies of the page image to produce.

Comments

This operator provides the most convenient mechanism for printing multiple copies of the same page image. With the exception of printing multiple copies of the page image, _showpages operates in exactly the same manner as showpage.

string

Description

Creates a string entirely consisting of ASCII NUL characters.

Usage

CharsInt **`string`** *NullStr*

CharsInt Integer. Number of characters to include in new string.

NullStr String. String containing *CharsInt* ASCII NUL characters.

Comments

The string operator automatically initializes the new string with ASCII NUL characters.

stringwidth

Description

Returns the current point relative movement which would occur if the user were to draw a given string using the show operator.

Usage

AnyStr **stringwidth** *XDeltaNum YDeltaNum*

AnyStr String. The string for which PAL will calculate the current point relative movement.

XDeltaNum Integer or fixed-point. Relative movement along the X axis.

YDeltaNum Integer or fixed-point. Relative movement along the Y axis.

Comments

The stringwidth operator allows a PAL procedure to calculate the distance a given string will cause the current point to advance after drawing the string using the show operator. The stringwidth operator does not actually draw the string. stringwidth performs the same current point movement calculations as show, but without drawing the string.

The operator returns the relative movement along both the X and Y axis. For Roman character set based languages, the interpreter will normally return zero for *YDeltaNum* and a positive value for *XDeltaNum*. However, other languages may draw their characters from left to right or vertically. These languages can produce a different range of possible values for *XDeltaNum* and *YDeltaNum*.

Comments

The stringwidth provides the means for a PAL procedure to automatically center or, in the case of Roman character set languages, right-justify text. The operator also provides the means for non-Roman character set languages to perform equivalent justifications.

The following PAL procedure will perform the same operation as the PAL show operator, however the procedure will automatically center the text over the current point.

```
/CShow {
  dup stringwidth
  exch 2 div exch 2 div rmoveto
  show
} bind def
```

The CShow procedure has the same usage as the show operator.

ShowStr **CShow**

For Roman text, the show operator draws *ShowStr* with the left bottom corner at the current point. The CShow procedure draws *ShowStr* with the center point of the bottom edge at the current point.

The following PAL procedure will perform the same operation as the PAL show operator, however the procedure will automatically reverse-justify the text against the current point. For Roman text, reverse-justification indicates right-justification.

```
/RShow {
  dup stringwidth
  exch neg exch neg rmoveto
  show
} bind def
```

The RShow procedure has the same usage as the show operator and CShow procedure.

ShowStr **RShow**

For Roman text, the RShow procedure draws *ShowStr* with the right bottom corner at the current point.

stroke

Description

Renders the current path onto the page.

Usage

```
stroke
```

Comments

Drawing lines onto a page involves a two step process. First, the programmer must specify a *path* consisting of the lines for PAL to draw. Second, the programmer uses the stroke operator to instruct PAL to draw the lines.

Operators like lineto and moveto add information to a path contained within the current graphics state. This path contains mathematical information concerning the drawing operations indicated by the programmer. A path can consist of several sub-paths. Each sub-path consists of a series of connected line segments. The moveto and closepath operators result in the termination of one sub-path and the start of a new sub-path.

The closepath operator allows the programmer to specify a *closed sub-path*. If the programmer terminates a sub-path using a moveto operator without a preceding closepath operator, PAL creates an *open sub-path*.

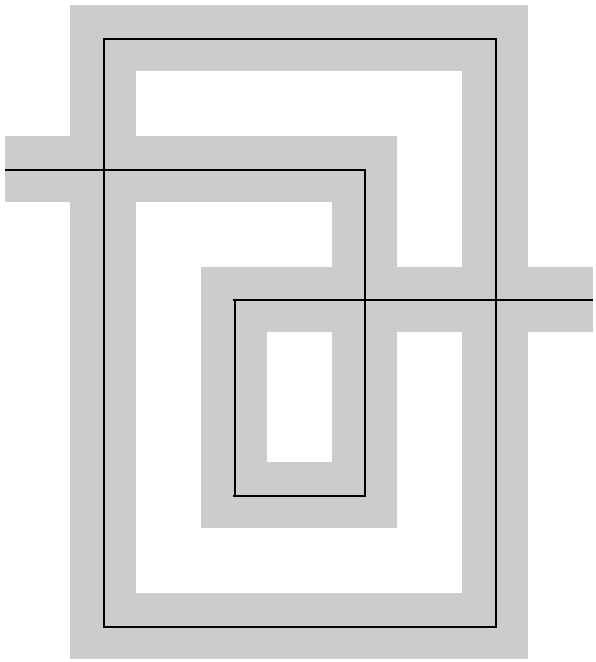
PAL joins together the starting and ending points of closed sub-paths and smoothes the transition between the connected line segments. For open sub-paths, PAL treats the start and end of the path as simple line end points.

The stroke operator instructs PAL to render all the current sub-paths onto the page using the current settings in the graphics state. The setlinewidth operator allows the programmer to alter the graphics state setting which controls the width of the lines drawn by stroke. For open sub-paths, setlinecap controls how PAL will render the start and end points of the sub-path.

Examples

The following example renders the diagram shown below the example. The example places the lower left corner of the diagram at user coordinate 36,36 and the upper right at 54,56.

```
% First build sub-path for loop which passes through box
  36 51 moveto
  47 51 lineto
  47 41 lineto
  43 41 lineto
  43 47 lineto
  54 47 lineto
% Next build sub-path for box which surrounds loop
  39 55 moveto
  51 55 lineto
  51 37 lineto
  39 37 lineto
  closepath
% Specify line width & end cap style, then stroke sub-paths
  2 setlinewidth
  0 setlinecap
  stroke
```



sub

Description

Subtracts the second number from the first number and returns the difference.

Usage

Any1Num Any2Num **sub** *DifNum*

Any1Num	Integer or fixed-point. First number from which to subtract the second number.
Any2Num	Integer or fixed-point. Second number to subtract from the first number.
DifNum	Integer or fixed-point. Integer if *Any1Num* and *Any2Num* are both integer, otherwise fixed-point. Result of *Any2Num* subtracted from *Any1Num*.

Comments

The sub operator pops the top two objects from the operand stack, subtracts the second from the first, and pushes the result back onto the operand stack. The interpreter must find two numeric objects on the top of the stack or a typecheck error will result.

If the stack contains two integer objects, the interpreter will perform integer subtraction and push an integer result onto the stack. The interpreter will perform fixed point subtraction and push a fixed point result if the stack contains a fixed point object as either operand.

translate

Description

Relocate the origin of the user coordinate system.

Usage

XTransNum YTransNum **`translate`**

XTransNum Integer or fixed-point. Distance by which to relocate the X axis in current user coordinates.

YTransNum Integer or fixed-point. Distance by which to relocate the Y axis in current user coordinates.

Comments

PAL applies the new translation to the current user coordinate system. Therefore, translations accumulate. The orders "2 4 translate 4 8 translate" produce the same affect as "6 12 translate".

trap

Description

Reserved operator for Applied Thermal Technologies internal use only.

truncate

Description

Eliminates the fractional portion of a given value.

Usage

AnyNum **`truncate`** *TruncatedNum*

AnyNum Integer or fixed-point. Value to truncate.

TruncatedNum Integer or fixed-point. Integer portion of *AnyNum*. The type of the returned value matches the type of the supplied parameter.

Comments

Although this operator will accept integer values, this operator has no affect upon integers. The following table shows the affect of the truncate operator upon various fixed-point values.

```
 1.6  ceiling  1.0
 1.5  ceiling  1.0
 1.4  ceiling  1.0
 1.0  ceiling  1.0
 0.0  ceiling  0.0
-1.0  ceiling -1.0
-1.4  ceiling -1.0
-1.5  ceiling -1.0
-1.6  ceiling -1.0
```

undef

Description

Removes an entry from a dictionary.

Usage

DictAny KeyAny **undef**

DictAny	Dictionary. Specifies the dictionary from which PAL should remove the entry.
KeyAny	Any. Specifies the key within the dictionary of the entry which PAL should remove.

Comments

PAL permits the selective deletion of entries within dictionaries. Given the dictionary and the key for an entry within the dictionary, the undef operator will delete the specified entry.

The programmer can use the undef operator in combination with the userdict operator to undefine objects previously defined using the def operator. See the userdict discussion for details.

PAL does not generate an error if *KeyAny* does not exist in the specified dictionary.

userdict

Description

Pushes the standard user dictionary, userdict, onto the top of the operand stack.

Usage

`userdict` *UserDict*

UserDict Dictionary. Standard user dictionary, userdict, from dictionary stack.

Comments

During initialization, PAL automatically creates three dictionaries -- systemdict, globaldict, and userdict. It then pushes these three dictionaries onto the *dictionary stack* in the listed order. Therefore, the dictionary stack contains systemdict at the bottom, globaldict in the middle, and userdict on the top.

PAL creates both userdict and globaldict as empty dictionaries. systemdict contains entries for all the names implicit to PAL. For example, the names of operators such as add, show, etc.

When PAL encounters an executable name object, it searches the dictionaries on the dictionary stack to locate the name. PAL starts with the dictionary on the top of the dictionary stack and proceeds down the stack until it finds an entry for the name. If PAL cannot locate the name in any dictionary on the stack, PAL generates an undefined error.

Normally userdict and globaldict do not contain names which match the names of PAL operators. Therefore, PAL does not find entries for the operator names in those dictionaries as PAL progresses down the dictionary stack. The interpreter does not find these operator names until it reaches the bottom of the dictionary stack. It then locates the operator name in systemdict. The systemdict entry for the operator name contains the intrinsic operator object which tells PAL which action to actually perform.

When the programmer uses the def operator, the programmer instructs PAL to add an entry to the user dictionary, userdict. Later, the programmer can instruct PAL to recall the information from userdict simply by supplying the name specified with the def operator. When PAL encounters the name, the interpreter searches the dictionary stack in the same manner as when PAL encounters the name of an operator. Since the dictionary stack contains userdict at the top, PAL immediately locates the programmer's entry in userdict.

When the programmer no longer requires an entry in userdict, the programmer should instruct PAL to remove the entry. PAL provides the undef operator for this purpose. However, the undef operator will work with any dictionary. This means the programmer must specify the dictionary upon which undef will operate. userdict provides the mechanism by which the programmer can undef entries created by def in userdict.

Examples

To remove an entry from userdict, the programmer need only use the operator userdict when specifying the dictionary upon which undef should operate. The following example defines the procedure DoNothing and then promptly undefines the procedure.

```
/DoNothing {1 2 add 3 sub pop} def
userdict /DoNothing undef
```

vmstatus

Description

Returns the number of bytes still available for allocation within the interpreter's virtual memory as well as the total size of the virtual memory.

Usage

`vmstatus` *ReservedInt UsedInt MaxInt*

ReservedInt Integer. Reserved for historical compatibility. Current release PAL interpreters always return zero for this value.

UsedInt Integer. Number of virtual memory bytes currently in use.

MaxInt Integer. Total virtual memory bytes within printer.

Comments

PAL stores all user data and procedures into a reserved area within the printer's memory called *virtual memory*. PAL refers to this memory space as virtual memory because the PAL programmer has only indirect access to the memory. PAL allows the programmer to store data and procedures into this memory space, however PAL controls the actual placement of this information into the virtual memory. The PAL programmer cannot directly manipulate the virtual memory area itself.

The PAL interpreter requires its own memory in order to save control information related to the operation of the printer as well as management information relating to the PAL programmer's data and procedures. Once PAL has allocated part of the printer's memory for this purpose, PAL allocates all of the remaining memory for use as the PAL virtual memory.

The vmstatus *MaxInt* value indicates the total size, in bytes, of the virtual memory area. This value will reflect the total amount of memory installed in the printer less the amount of memory the PAL interpreter requires.

The *UsedInt* value indicates the total number of bytes which the PAL programmer currently has in use within the virtual memory area. Even when the programmer has no data or procedures stored within the printer, *UsedInt* will still show virtual memory in use by the operand stack and various other PAL language data structures. Since PAL dynamically grows and shrinks data areas like the operand stack to meet the user's requirements, PAL allocates these data areas within virtual memory.

Hints

The PAL sequence "vmstatus exch sub exch pop" will leave on the top of the stack the total number of bytes still available for allocation within virtual memory. "exch sub" exchanges *UsedInt* and *MaxInt* on the top of the stack and then subtracts *UsedInt* from *MaxInt* to determine the numbers of bytes not used. "exch pop" then exchanges the result with *ReservedInt* on the top of the stack and discards *ReservedInt* from the stack. This leaves only the number of bytes available on the top of the stack.

PAL manages the virtual memory space in an extremely dynamic manner. In addition, the size of PAL data objects can vary from printer to printer. Therefore, the programmer should use the

vmstatus operator to determine the amount of space necessary to store a given amount of data within a given printer.

To determine the amount of memory necessary to store a given amount of data, the programmer should use vmstatus to determine a starting memory usage level. The programmer should then download a representative sample of the data to the printer, and check the new memory status using the vmstatus operator.

The PAL operator includes numerous data management optimizations. Therefore, a very small amount of data may not provide a true representation of memory usage. In general, the programmer should probably download sufficient data to the printer to create at least a 64,000 byte change in the value reported by vmstatus.

writestring

Description

Write data to an open file.

Usage

OpenFile AnyStr **`writestring`**

OpenFile	File. A file object returned by the file operator. The programmer must have opened the file for writing.
AnyStr	String. A string containing the data to write to the file.

Comments

PAL will write the data contained in the string to the indicated file without any modification.

xor

Description

Performs a logical or bit-wise exclusive-or operation on two boolean or integer values.

Usage

Any1Bool Any2Bool **xor** *XorBool*
Any1Int Any2Int **xor** *XorInt*

Any1Bool Boolean. First operand for the logical exclusive-or operation.

Any2Bool Boolean. Second operator for the logical exclusive-or operation.

XorBool Boolean. Result of the logical exclusive-or operation.

Any1Int Integer. First operand for the bit-wise exclusive-or operation.

Any2Int Integer. Second operand for the bit-wise exclusive-or operation.

XorInt Integer. Result of the bit-wise exclusive-or operation.

Comments

The following table lists the results of performing the logical exclusive-or operation on two boolean values.

		Any1Bool	
		false	true
Any2Bool	false	false	true
	true	true	false

The following table lists the results for each bit position when performing the bit-wise exclusive-or operation on two integer values.

		Any1Int	
		0	1
Any2Int	0	0	1
	1	1	0

A. Bar Code Considerations

Precision Bar Code Control

PAL's philosophy allows you to run the same PAL program on different printers and produce labels that look the same, regardless of the printing method and printer resolution. PAL performs this function very well for letters, lines, and boxes. There are, however, physical constraints that prevent PAL from exactly reproducing bar codes on printers with different resolutions. PAL will always attempt to adapt the bar codes to different printers and, although the bar codes will not match exactly, they will usually be acceptable.

If you must have precise control, your PAL programs will become machine dependent. In operational situations, this is not usually a problem. Getting precise control over bar code appearance is not difficult. The major constraint that PAL must work with while converting bar code dimensions from user to device coordinates is *bar code dimensions must be an integral number of dots*. There is no way to print a fraction of a dot.

To illustrate how this might be a problem, consider what happens when a PAL program requests a Code-39 bar code with a NarrowWidth of 0.01" and a WideRatio of 2.7:1 to be printed on a printer with 8 dots/mm (203 dots/inch). The first problem comes with the conversion of 0.01" to dots. To make narrow bars exactly 0.01" wide would require 2.03 dots. The best PAL can do on the 8 dots/mm printer is 2 dots. In this case, the error is probably not noticeable.

However, look what happens when the width of the wide bars is calculated by multiplying those 2 dots by 2.7. The result is 5.4 dots. PAL will round this to 5 dots, giving an effective WideRatio of 2.5:1. There is no way to avoid this problem. If precise WideRatios are required, careful selection of the NarrowWidth with a knowledge of the print density is required. A better solution is to choose a WideRatio that can be achieved at the current resolution (for this example, 2:1, 2.5:1, or 3:1).

Now suppose bar codes made with the above program on the 8 dot/mm printer was acceptable as PAL made it. What happens when the same program is run on a printer with 12 dots/mm (305 dots/inch)? The NarrowWidth will be 3 dots (with an insignificant error of .05 dots/bar). The wide bars will be 8 dots wide, giving an effective WideRatio of 2.67:1. This will give wider bar codes than the 8 dot/mm case and the difference would be noticeable. However, both bar codes would probably be acceptable.

To take control of the bar code dimensions, you must work in dots instead of points, inches, or millimeters. Doing this will, however, reduce the portability of your PAL programs. The following PAL program will produce narrow bars exactly 2 dots wide with a ratio of 2.5:1 on an 8 dots/mm printer. Narrow bars will be 0.0099" (0.250 mm) wide and the wide bars will be 0.0246" (0.6256 mm) wide.

```
/dots {203 div 72 mul} bind def
/bar {
  <<
    /NarrowWidth 2
    /WideRatio 2.5
    /CheckDigit true
  >>
  /Code39 _barcode
} bind def
72 72 moveto
(CODE39) bar
showpage
```

The dots procedure converts dots for an 8 dot/mm print density to points.

Bar Code Parameter Defaults

In the discussion of the individual bar code commands, default values for various parameters were given. All dimensioned values had a parenthetical note giving the dimensions in inches for the default current transformation matrix (CTM). You should note that these numbers are invariant with respect to the transformation matrix. This means that if the default for a given dimension is used, you will only get the expected results if the user units are points. If, for instance, Height is set to 36, the bar code will be 0.5 inches high using the default CTM. If, however, you have scaled the user units so that you are working in inches, the Height is still 36. Now, however, it will be 36 inches, making a very tall bar code.

There are several solutions to this problem. The easiest solutions are

1) always work in points (don't change the CTM) or

2) explicitly specify all the bar code dimensions, thus overriding the defaults.

A handy way to work in points, but still be able to specify individual dimensions in inches or millimeters, is to define simple procedures to locally convert the dimensions. The following example will specify the height of a Code-128 bar code in millimeters, place the bar code at a specific location in inches, and use the default value for NarrowWidth.

```
/inches {72 mul} bind def
/mm {25.4 div 72 mul} bind def
1.0 inches 1.5 inches moveto
(~bCode-128) <</Height 20 mm>> /Code128 _barcode
showpage
```

Determining the Width of Bar Code Bit Maps

It is sometimes necessary to compute the width of the bit map generated by the _barcode function. The following formulas will allow you to estimate this width. For the greatest precision, the calculations should be done in dots (see discussion above).

The formulas use the following variables:

L	The width of the bar code bit map.
X	Width of narrow elements in the symbol. The units of this variable determine the units of the answer (*i.e.*, if X is in dots, L will be in dots).
N	The number of characters including function and shift codes, if applicable, and excluding characters covered by D, P, and W.
D	Number of numerical digits in Code 128 mode C.
P	Number of partial or complete pairs of digits in I 2 of 5 bar codes.
W	Number of : / . and + characters in Codabar symbols.
R	Wide to narrow ratio of elements in symbologies that have wide and narrow elements.
Q	Quiet zone width. Number of narrow elements (X) to leave blank at the beginning and end of the symbol. This value is always 10 in PAL symbols.

Use the following formulas to estimate the bar code width:

Code 39	L = N + 1 + (N + 2) * (6 * X + 3 * R * X) + 2 * Q
Code 93	L = ((4 + N) * 9 + 1) * X + 2 * Q
Code 128	L = ((5.5 * D + 11 * N + 35) * X) + 2 * Q
CODABAR	L = (((2 * R + 5) * N + (R - 1) * (W + 2) + (N - 1) * X) + 2 * Q
EAN-13	L = 95 * X + 2 * Q
EAN-13+2	L = 125 * X + 2 * Q
EAN-13+5	L = 153 * X + 2 * Q
EAN-8	L = 67 * X + 2 * Q
I 2 of 5	L = (P * (4 * R + 6) + 6 + R) * X + 2 * Q
UPC-A	L = 95 * X + 2 * Q
UPC-A+2	L = 125 * X + 2 * Q
UPC-A+5	L = 153 * X + 2 * Q
UPC-E	L = 51 * X + 2 * Q

Note that all PAL bar code bit maps have a 10X quiet zone at each end. If you need to place the bar portion of the bit map at a specific location, start the bar code 10X to the left of the desired location. Be sure to leave at least 10X of white space or the bar code may not scan properly. Also, remember that UPC and EAN symbols print human-readable text in the quiet zone.